NAVAL POSTGRADUATE SCHOOL
Monterey, California

THESIS

A FIXED-POINT PHASE LOCK LOOP IN A SOFTWARE DEFINED RADIO

by

Michael T. Johannes

September, 2002

Thesis Advisor:	Tri Ha
Second Reader:	Roberto Cristi

THIS PAGE INTENTIONALLY LEFT BLANK

REPORT DOCUMENTATION PAGE			Form Approved OMB No. 0704-0188
Public reporting burden for this collection of information is estimated to average 1 hour per response, including the time for reviewing instruction, searching existing data sources, gathering and maintaining the data needed, and completing and reviewing the collection of information. Send comments regarding this burden estimate or any other aspect of this collection of information, including suggestions for reducing this burden, to Washington headquarters Services, Directorate for Information Operations and Reports, 1215 Jefferson Davis Highway, Suite 1204, Arlington, VA 22202-4302, and to the Office of Management and Budget, Paperwork Reduction Project (0704-0188) Washington DC 20503.			
1. AGENCY USE ONLY ***(Leave blank)***	**2. REPORT DATE** Month Year	**3. REPORT TYPE AND DATES COVERED** Master's Thesis	
4. TITLE AND SUBTITLE: Title (Mix case letters) A Fixed-point Phase Lock Loop in a Software Defined Radio			**5. FUNDING NUMBERS**
6. AUTHOR(S)			
7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES) Naval Postgraduate School Monterey, CA 93943-5000			**8. PERFORMING ORGANIZATION REPORT NUMBER**
9. SPONSORING /MONITORING AGENCY NAME(S) AND ADDRESS(ES) N/A			**10. SPONSORING/MONITORING AGENCY REPORT NUMBER**
11. SUPPLEMENTARY NOTES The views expressed in this thesis are those of the author and do not reflect the official policy or position of the Department of Defense or the U.S. Government.			
12a. DISTRIBUTION / AVAILABILITY STATEMENT Approved for public release; distribution is unlimited.			**12b. DISTRIBUTION CODE**

13. ABSTRACT ***(maximum 200 words)***

A software defined radio is a much more flexible platform than traditional, hardware implemented radios. By implementing radio functions in software, and putting those functions on a Field Programmable Gate Array (FPGA) chip, users will have the ability to download mission specific radio capabilities. This thesis examines a fundamental piece of the receiver, the Phase-Lock Loop (PLL), simulates a software PLL, and investigates the effects of fixed-point versus floating point mathematics required for an FPGA based PLL. With a fixed-point PLL simulator, figures of merit such as lock-time, lock range, and pull-in range are determined for typical signal-to-noise ratio (SNR) levels

14. SUBJECT TERMS software radio, fixed-point, phase lock loop, field-programmable gate array			**15. NUMBER OF PAGES** 89
			16. PRICE CODE
17. SECURITY CLASSIFICATION OF REPORT Unclassified	**18. SECURITY CLASSIFICATION OF THIS PAGE** Unclassified	**19. SECURITY CLASSIFICATION OF ABSTRACT** Unclassified	**20. LIMITATION OF ABSTRACT** UL

NSN 7540-01-280-5500 Standard Form 298 (Rev. 2-89)
Prescribed by ANSI Std. 239-18

THIS PAGE INTENTIONALLY LEFT BLANK

Approved for public release; distribution is unlimited

A FIXED-POINT PHASE LOCK LOOP IN A SOFTWARE DEFINED RADIO

Michael T. Johannes
Captain, United States Marine Corps
B.A.,College of the Holy Cross, 1995

Submitted in partial fulfillment of the
requirements for the degree of

MASTER OF SCIENCE IN ELECTRICAL ENGINEERING

from the

NAVAL POSTGRADUATE SCHOOL
September 27, 2002

Author: Michael T. Johannes

Approved by: Tri Ha
Thesis Advisor

Roberto Cristi
Second Reader/Co-Advisor

John Powers
Chairman, Electrical and Computer Engineering Department

THIS PAGE INTENTIONALLY LEFT BLANK

ABSTRACT

A software defined radio is a much more flexible platform than traditional, hardware implemented radios. By implementing radio functions in software, and putting those functions on a Field Programmable Gate Array (FPGA) chip, users will have the ability to download mission specific radio capabilities. This thesis examines a fundamental piece of the receiver, the Phase-Lock Loop (PLL), simulates a software PLL, and investigates the effects of fixed-point versus floating point mathematics required for an FPGA based PLL. With a fixed-point PLL simulator, figures of merit such as lock-time, lock range, and pull-in range are determined for typical signal-to-noise ratio (SNR) levels.

THIS PAGE INTENTIONALLY LEFT BLANK

TABLE OF CONTENTS

I. INTRODUCTION 1
A. BACKGROUND 1
1. The Phase-Lock Loop 2
2. Fixed-Point Arithmetic 2
B. OBJECTIVES 4
C. RELATED RESEARCH 4
1. Digital Delay Lock Loop 4
2. Digital Downconversion and Channelization 5
D. THESIS ORGANIZATION 5

II. PHASE LOCK LOOP BASICS 7
A. PRINCIPLES OF THE PHASE-LOCK LOOP 7
1. Phase Detector (PD) 8
2. Loop Filter 9
3. Voltage Controlled Oscillator / Numerically Controlled Oscillator 10
B. OPERATION OF THE PHASE LOCK LOOP 11
1. The Transfer Function 13
a. Phase Step Applied to Input 15
b. Frequency Step Applied to Input. 16
c. Frequency Ramp Applied to Input 17
C. PERFORMANCE MEASURES OF THE PLL 17
1. The Lock Range 18
2. Lock Time 19
3. The Pull-in Range 19
4. Pull-in Time 19
5. The Pull-out Range 19
D. PERFORMANCE OF THE PLL IN NOISE 21

III. FIXED-POINT ARITHMETIC 25
A. FLOATING-POINT NUMBERS 25
B. FIXED-POINT NUMBERS 26
1. Fixed-point Number Representation 26
2. Precision and Range in Fixed-point Arithmetic 28
3. Errors in Fixed-point Numbers 29

IV. FIXED-POINT PLL SIMULATION 31
A. PROCEDURE FOR MODEL 31
1. Floating-point MATLAB Simulation 31
a. Input Signal Assumptions 32
b. Determination of PLL Parameters 32
c. The MATLAB Simulation 35
2. Fixed-point Simulink Model 38
3. Fixed-point Model 39
B. PERFORMANCE AND ANALYSIS OF SIMULINK PLL MODEL 41

1. Performance Measures of the Fixed-point Model 41
a. Lock Range 43
b. Lock Time 43
c. Pull-in Range 44
d. Pull-in Time 44
e. Pull-out Range 45
f. Frequency drift 45
2. Analysis of PLL 48
a. Errors in the Output of the PLL 48
b. The 8-bit PLL 54
c. Effects of Changing PLL Parameters 55
E. CONVERSION TO HARDWARE 55
V. CONCLUSIONS 57
A. CAPABILITIES AND LIMITATIONS 57
B. RECOMMENDATIONS FOR FURTHER RESEARCH 58
C. FINAL COMMENTS 59
APPENDIX A 61
LIST OF REFERENCES 69
INITIAL DISTRIBUTION LIST 71

LIST OF FIGURES

Figure 1. Ideal Software Defined Radio [3] xiii
Figure 2. Simple Phase-Lock Loop xv
Figure 3. Phase-Lock Loop with Signals of Interest 7
Figure 4. Bode Plot of Active PI Filter. (From Ref [1]) 10
Figure 5. Exciting Functions as Applied to Input of a PLL. (a) Phase Step. 11
(b) Frequency Step. (c) Frequency Ramp. (From Ref [1]) 11
Figure 6. Block Diagram of PLL Transfer Function 14
Figure 7. Time Response of PLL for different values of ζ and $\omega_n = 200$. 16
Figure 8. Relationship of frequency ranges of a PLL. (From Ref. [1].) 20
Figure 9. T_{av} plotted as a function of SNR_L, where T_{av} is normalized to the natural frequency. (From Ref. [2].) 22
Figure 11. (a) Frequency Time Response of PLL. (b) Phase Error Time Response of PLL. 37
Figure 12. Time Response of PLL for an Input Frequency Larger than the Lock Range. (a). Output of System Demonstrating the Signal Locks on a Frequency of 2350 Hz. (b). Plot of the Error of the System. 38
Figure 13. (a). Plot of Fixed-point Model NCO Output. (b). Plot of Fixed-point Model Error. 42
Figure 14. Vehicle Driving 60 mph Around a Tight Curve to Illustrate how the Doppler Shift can Cause a Frequency Ramp of a Signal. 47
Figure 15. Error Plot of the Output of the Fixed-point NCO versus Time 53
Figure A1. Phase-Lock Loop Simulink Model 64
Figure A2. Phase Detector Simulink Model 65
Figure A3. Simulink Active P-I Loop Filter 66

THIS PAGE INTENTIONALLY LEFT BLANK

LIST OF TABLES

Table 1. Range and Precision of a 16-bit Fixed-point Data Type 29

THIS PAGE INTENTIONALLY LEFT BLANK

EXECUTIVE SUMMARY

The explosion of wireless and PCS services over the last decade has created numerous incompatible air interface standards. A subscriber to one service will find his phone rendered useless when roaming in the coverage area of another service, using a different standard. These competing transmission formats each might have their own unique modulation type, multiple access technique, error control methods, call set-up and handoff protocol and voice compression algorithms. The military has seen similar interoperability of radio standards between tactical radios used by coalition forces in Desert Storm.

The need for a flexible communication platform, capable of interfacing with the numerous standards and formats has become apparent. A software radio, implementing traditional radio functions in software, gives the user this flexibility. The ideal software radio shown in Figure (1) would digitize the entire received signal spectrum using a high speed Analog-to-Digital Converter (ADC), perform all demodulation, data protocol and processing functions using a general-purpose digital signal processor (DSP).

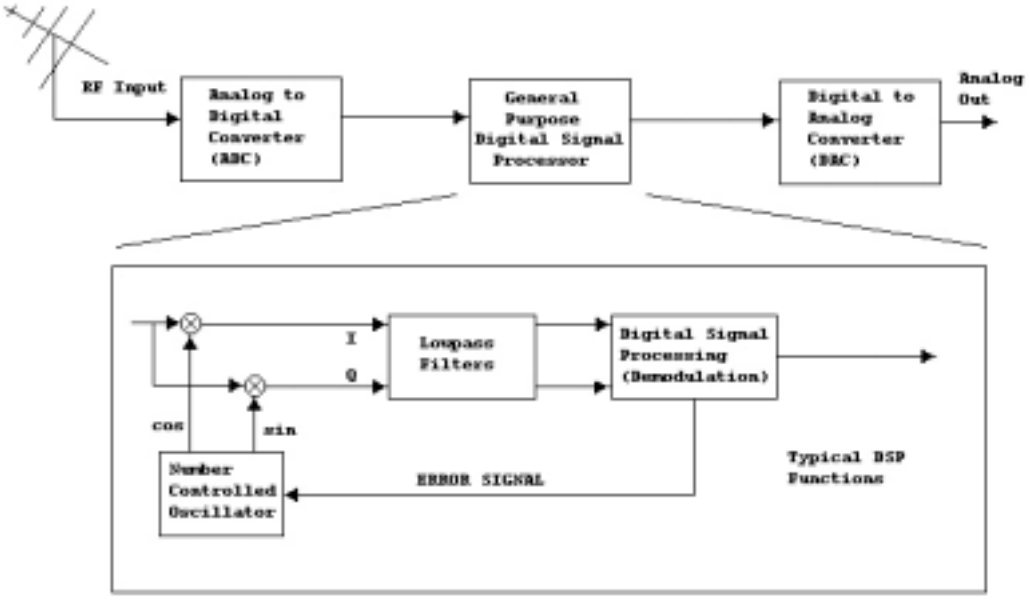

Figure 1. Ideal Software Defined Radio. (From Ref. [3].)

Due to limitations on A/D conversion speed, sampling the direct RF spectrum is not typically an option. A practical software radio will usually incorporate an RF front end, which filters and downconverts a portion of the received signal spectrum to much lower IF frequency such as 21.4, 70 or 160 MHz. These frequencies can be digitized directly with current state of the art A/D converters. This technique is called "IF Sampling".

The digital signal processing (DSP) functions of the software radio can be implemented either with general-purpose DSP chips or with reconfigurable Field Programmable Gate Arrays (FPGAs). In either case, the processing functions are in the form of software, available for download to the DSP engine. The resulting unit would have the capability to be reconfigured for any radio signal format. This gives the user the ability to download mission specific radio requirements, using the same platform for numerous radio applications.

This thesis will focus on the implementation issues for an important signal processing function common to most communications receivers; that is, the Phase-Lock Loop (PLL). The PLL can take several forms such as the Costas Loop for carrier recovery and tracking, the early-late gate synchronizer for baud timing recovery, and the delay-locked loop (DLL) for spreading sequence tracking in spread spectrum systems. In each case, the same PLL loop theory presented in this thesis applies.

In short, the PLL is a feedback loop device, which locks onto a received signal, meaning it synchronizes its output in-phase and frequency with its input. The PLL can be broken down into its three component parts: 1.) the phase detector (PD), 2.) a loop lowpass filter, and 3.) a voltage controlled oscillator (VCO) or numerically controlled oscillator (NCO), the latter being used in a software PLL version. Figure 2.shows a simple PLL schematically.

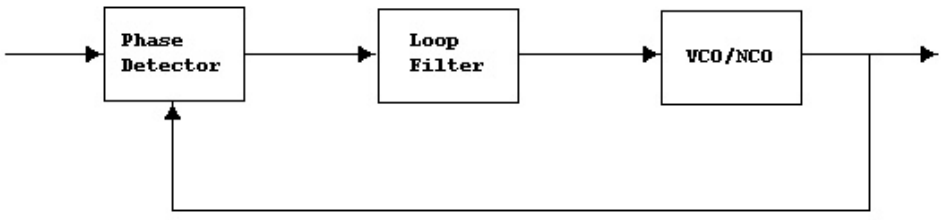

Figure 2. Simple Phase-Lock Loop

While a traditional analog receiver implements the PLL components exclusively in hardware, a software-defined radio requires these components to be available as downloadable software, using a digitized signal as the input. While a software simulation of a PLL has been a reality for some time, the advent of fast Field Programmable Gate Array (FPGA) technology makes this a useful concept for high data rate signals (those greater than roughly 2 Mbit/sec data rate). Currently, the software components can be programmed onto an FPGA and used real-time for DSP functions.

When deciding on DSP implementations, one must consider whether fixed-point or floating point arithmetic and number representation will be used. The benefit of floating point is the large dynamic range associated with the floating-point number representations. When implementing algorithms in floating point, the designer typically does not have to worry about issues such as rounding or truncation error, or numeric overflow. The disadvantage of floating point is the increased computational resources required and the processing speed limitations. For this reason, floating point implementations of receiver algorithms on general purpose DSP are limited to relatively low data rate signals.

One of the reasons FPGAs are fast enough to be used is the fact that they use fixed-point representations of numbers rather than floating point. Fixed-point representation is a much more efficient way for a computer to do arithmetic, because its essentially a binary representation of a decimal number. Hence, a computer can do

arithmetic at a much faster rate using fixed-point numbers. With the computationally heavy nature of any DSP application, the efficiencies of fixed-point arithmetic add up to significant savings in time. The trade off is that representing numbers in this way either reduces the range of the number, or its precision. As an example, using 8-bit fixed-point numbers, a range of –256 to 256 only has a precision of 2.0. For a precision of .01, the range of numbers available are –2 to 2. This constraint needs to be realized when programming using fixed-point arithmetic.

The aim of this research is to model a software phase-lock loop, observe its performance, and convert it into a fixed-point implementation to determine the effects on important performance such as lock time and pull-in range. The questions that required answering were

- How many bits are needed in a fixed-point implementation for acceptable performance?
- What type of errors do a fixed-point implementation introduce into the output signal?
- What effect does fixed-point arithmetic have on figures of merit of a PLL?
- What kind of signal-to-noise ratios (SNR) are required to lock the PLL in an acceptable time

The source code used initially was MATLAB, but to convert the model into fixed-point, the MATLAB extension Simulink was used. The final simulation model used 16-bit fixed-point arithmetic and locked with acceptable SNR's, with only small errors to the output signal. Lock time and ranges were not changed, however a smaller pull-in range than the floating-point equivalent was encountered. Errors at the output of the PLL due to the fixed-point implementation are observed, but nothing of critical size. The fixed-point model performed comparable to an analog or floating-point model in all ways, and could used as a basis to build a PLL on a FPGA or DSP chip.

I. INTRODUCTION

A. BACKGROUND

The revolution in wireless technology in the last decade has created a need for a fast, flexible, and light radio, compatible with the numerous transmission formats and standards. The concept of a software radio has been developed to fill that need. The ideal software radio digitizes the radio spectrum at the receive antenna, providing all demodulation, decryption, and signal processing in software. Such a tool would have the capability to download appropriate software, depending on the mission and the transmission format required. It could be reconfigured to accommodate any RF-band modulation or data format or transmission standard, capable of operating within any communications network.

To realize the ideal software radio, the entire signal spectrum would be digitized at the antenna by an analog-to-digital converter (ADC). A practical SW radio will usually incorporate an RF front end which filters and downconverts a portion of the received signal spectrum to much lower IF frequency which can be digitized directly with current state of the art A/D converters. The digital signal processing (DSP) is then performed in software in a reprogrammable Field Programmable Gate Array (FPGA). Finally, the demodulated signal is sent through a digital-to-analog converter (DAC) to generate the audio or video output if required. The ideal software radio architecture is shown in Figure 1, where the DSP functions are done on an FPGA chip. The radio concept described above could be reprogrammed to accommodate any radio standard or air interface by downloading the appropriate software algorithm.

The signal processing functions required by the software radio includes tuning, filtering, demodulation and decryption. Tuning is accomplished by mixing the digitized signal with a digital local oscillator to down-convert the signal to baseband. This mixer is simply a signed multiplier, sample by sample in the digital implementation. Typically, a complex baseband signal representation is used. The mixer will output the in-phase and quadrature-phase components in sine and cosine waveforms. This mixing process creates unwanted frequencies, specifically at twice the receive frequency. This spectral component is removed using a finite impulse response filter (FIR). The digital filtering also bandlimits the samples to the bandwidth of the signal of interest (a process called “channelization”).

Demodulation is performed by a variety of methods, depending on the modulation type. This research assumes demodulation will be performed by phase locking to the incoming signal. An error signal is fed back to the tuner to adjust the local oscillator frequency in a closed loop system. This feedback system is simply a digital or software phase-lock loop (PLL). The implementation of such a PLL is the focus of this research.

1. The Phase-Lock Loop

A simple phase-lock loop is pictured in Figure 2 and can be seen to consist of three component parts: 1.) the phase detector, 2.) a loop filter, and 3.) a voltage controlled oscillator (VCO) or numerically controlled oscillator (NCO). Its purpose is to lock on to the frequency and phase of the input signal. In this case, the purpose of the PLL is to create a phase coherent as a local oscillator in the receiver for demodulation.

The phase detector or phase comparator compares the phase between the input signal and the output signal. It generates a signal proportional to the phase error, or difference between the two signals phase. This can be done numerous ways; the simplest is to multiply the two signals together.

The loop filter block is a low-pass filter that removes the high frequency terms the come from the multiplication of the input and output signal, leaving only the phase error.

The NCO takes the phase error from the loop filter output and adjusts its output sinusoidal signal to force the error to zero. This adjusted signal is the feedback signal that goes into the phase detector, producing a second phase error and the process repeats.

This type of PLL is a second-order system. The error eventually settles to zero, but the output is a damped oscillation. This oscillation is governed by the parameters of the PLL.

2. Fixed-Point Arithmetic

Measurements of physical quantities can take on many numerical representations. For example, the number one thousand can be represented by 1000, 1E3, 10^3, or one thousand. In this case, the same quantity is represented using four different syntaxes. Another example of the same

quantity, represented in different *scales,* is the boiling point of water. Water boils at 100 degrees Celsius, 212 degrees Fahrenheit, 373 degrees Kelvin, or 671.4 degrees Rankine.

Determining an appropriate scale and representation depends on many factors. Suppose you need to measure the speed of a vehicle. The numerical values have a limited range for this application. The slowest it can travel is 0 mph at a dead stop and the top speed of the vehicle has been determined to be 150 mph. If an 8-bit unsigned integer is required, values in the range of 0-255 are possible.

A typical approach would be to assign one bit per mile-per-hour, making the integer 0 a dead stop and 150 the vehicle's top speed. This scheme, while easy to convert, neglects the use of the integers 151-255, wasting 40% of the number range.

Another approach would be to set the integer zero to a dead stop and the integer 255 to the top speed. This scale gives us much greater precision, 0.58823 mph per bit, because all 256 values are used in the vehicle speed. The conversion, however, requires a division of 1.7, a relatively expensive operation for fixed-point processors.

The trend of recent technology is to implement control systems and digital signal processing functions on digital hardware. Within digital hardware, numbers are represented as either floating-point or fixed-point data types. The number of bits used to represent both data types is a fixed word size. The range of fixed-point representation is much smaller than for floating point, thus to avoid overflow and quantization errors, fixed-point representation must be scaled. If floating-point numbers can effectively represent real world values with virtually no error, why use a fixed-point based implementation? The answer, of course, is cost, size and processing speed.

A fixed-point hardware platform is architecturally much simpler than its floating-point counterpart. This means cheaper manufacture of the product. In addition to manufacturing savings, if scaled properly, fixed-point arithmetic can be significantly faster, saving computation time.

Because the logic circuits of fixed-point hardware are less complicated than that of floating point, the chip size can be much smaller, reducing power consumption. This means

smaller batteries, and reduced heat as a by-product, removing the need for an expensive and bulky heat sink.

For a software radio, programmed on a FPGA where speed is a critical limitation, and size and battery life are a concern, a fixed-point implementation is the only plausible solution.

B. OBJECTIVES

This research focused on the effects of fixed-point arithmetic on a simulated PLL, specifically, what number of bits are required to successfully implement a PLL, how does this affect the figures of merit of a PLL such as pull-in range, lock time, and lock range, and to determine the signal-to-noise ratios (SNR) needed for a fixed-point PLL to be effective in a software radio.

These objectives were accomplished using MATLAB's Simulink software. A PLL simulation using only fixed-point arithmetic was implemented, taking as input a noisy signal. This simulation was done using 16-bit fixed-point arithmetic. Simulation results were consistent with theoretical results, with the fixed-point implementation simply adding "noise" to the signal, causing the output signal to have small errors due to the quantization effects of fixed-point arithmetic.

C. RELATED RESEARCH

The related research is in identifying the feasibility and requirements for other components to make a software radio realizable. Some of this research is being done concurrently with this research. The first is a digital delay lock loop (DDLL); the second is investigating digital downconversion and channelization.

1. Digital Delay Lock Loop

For a fully functional software radio, a real-time, spread-spectrum, signal-processing block is desired. An integral function of a direct sequence spread spectrum receiver is recovering the underlying narrowband data through a procedure called dispreading. The key component of dispreading is a DDLL. A DLL generates an exact replica of the spreading sequence generated at

the transmitter, allowing the recovery of the narrowband data. This research was being conducted by Captain Samuel Laboy, USMC.

2. Digital Downconversion and Channelization

A basic building block of any all-digital receiver is the downcoversion of a real signal at an intermediate frequency to complex in-phase and quadrature-phase components at baseband. The downcoversion component of the software radio consists of a complex NCO, local oscillators that act as mixers, and digital filter. Investigation in the feasibility of digital downconversion and performance characteristic compared to an analog equivalent is being pursued by Lieutenant Michael Snelling, USN. The aim of this research is to build a working MATLAB model and to analyze the results of the simulation.

D. THESIS ORGANIZATION

This thesis is organized to mirror the research. Chapter 2 is an overview of the operation of the PLL and derives the figures of merit for analysis of the final model. Chapter 3 is a tutorial on fixed-point arithmetic and explains the advantages that it offers over floating-point arithmetic. Chapter 4 is the culmination of this research. It begins by describing the building of a floating-point PLL and then the procedures for converting it to a fixed-point model in Simulink. Analysis of the performance of the fixed-point PLL according to the derived figures of merit derived in Chapter 2 is also done in Chapter 4. The conclusions for the feasibility of a FPGA based PLL and recommendations for future research are in Chapter 5. The Appendix gives the MATLAB code and Simulink model for the floating-point and fixed-point phase lock loops.

THIS PAGE INTENTIONALLY LEFT BLANK

II. PHASE LOCK LOOP BASICS

A. PRINCIPLES OF THE PHASE-LOCK LOOP

A phase-lock loop is a circuit, or software, designed to track a given Reference signal in both frequency and phase. Its applications are far reaching from AM/FM radio demodulation, to television sets, to coders and decoders. The PLL has the ability to synchronize its output in-phase and frequency with an input signal, meaning the phase error between the PLL's output and the input signal is zero, or remains constant. If a phase error is introduced, by a phase change or a frequency change, the PLL's feedback control mechanism adjusts the oscillator's output to account for it.

To get a basic understanding of the PLL, the principles of the linear PLL will be examined, which is shown schematically in Figure 3. It consists of three blocks: the phase detector (PD), the loop filter (LF), and the VCO/NCO. For ease of Reference, the same notation and signal names used by [1] will be used here. Consequently the frequency of $u_1(t)$ in radians/second is ω_1, ω_2 is the frequency of $u_2(t)$ and $\theta_e(t)$ is the phase error or difference in-phase of $u_1(t)$ and $u_2(t)$. With signals defined, a closer look at each functional block is in order.

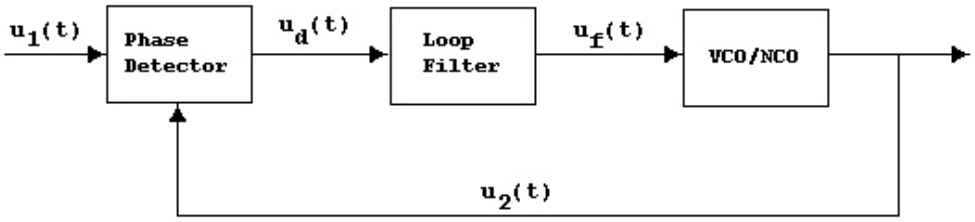

Figure 3. Phase-Lock Loop with Signals of Interest

1. Phase Detector (PD)

The PD takes as its input two signals, the reference signal, $u_1(t)$, and the output of the VCO, $u_2(t)$. This block simply compares the phases of these two signals and produces a signal $u_d(t)$ proportional to the phase error θ_e, specifically,

$$u_d(t) = K_d\theta_e \tag{2.1}$$

where K_d is the PD gain in volts. Obtaining the signal $u_d(t)$ can be as simple as a multiplier. The PD adopted in the simulation uses the in-phase and quadrature-phase portions of the input and output signal. To see how this works, lets assume a phase difference between the $u_1(t)$ and $u_d(t)$ of θ_e. Defining the in-phase and quadrature-phase components as $I_1 = U_1\cos(\omega_1 t)$, $Q_1 = U_1\sin(\omega_1 t)$, $I_2 = U_2\cos(\omega_2 t + \theta_e)$, and $Q_2 = U_2\sin(\omega_2 t + \theta_e)$, so that

$$\begin{aligned} u_1(t) &= U_1(\cos(\omega_1 t) + i\sin(\omega_1 t)) \\ u_2(t) &= U_2(\cos(\omega_2 t + \theta_e) + i\sin(\omega_2 t + \theta_e)) \end{aligned}$$

where U_1 and U_2 are the amplitude of the respective signals and is related to the PD gain K_d. Using the trigonometric identities

$$\cos x\cos y = \frac{1}{2}[\cos(x+y) + \cos(x-y)] \text{ and } \sin x\sin y = \frac{1}{2}[\cos(x-y) - \cos(x+y)]$$

and a little algebra, θ_e can be extracted. For ease of computation, we assume $\omega_1 = \omega_2$. If this is not the case, the PD output has an additional frequency error, but this ac component will be filtered out by the loop filter explained below.

$$\begin{aligned} I &= I_1 * I_2 + Q_1 * Q_2 \\ &= U_1 * U_2[\cos(\omega_1 t)\cos(\omega_2 t + \theta_e) + \sin(\omega_1 t)\sin(\omega_2 t + \theta_e)] \\ &= \frac{1}{2}U_1 * U_2[\cos((\omega_1 + \omega_2)t + \theta_e) + \cos(\theta_e) + \cos(\theta_e) - \cos((\omega_1 + \omega_2)t + \theta_e)] \\ &= U_1 * U_2\cos(\theta_e). \end{aligned} \tag{2.2}$$

Similarly,

$$\begin{aligned} Q &= I_1 * Q_2 - Q_1 * I_2 \\ &= U_1 * U_2 \sin(\theta_e). \end{aligned} \tag{2.3}$$

Using equations (2.2) and (2.3),

$$\theta_e = \tan^{-1}\left(-\frac{Q}{I}\right). \tag{2.4}$$

As desired, the PD output signal $u_d(t)$ is the phase error.

2. Loop Filter

As alluded to earlier, if the frequencies of the two signals differ or alternate PD implementations are used, the output signal, $u_d(t)$, of the PD will have an unwanted ac component. This ac signal is superimposed on the desired dc component representing the phase error. To remove the ac component, a simple low pass loop filter is used.

To implement a loop filter, numerous strategies exist. While a high-order finite impulse response (FIR) filter will ensure the ac component is removed, the delay in response is too great, where locking onto a signal as fast as possible is needed. A first-order low-pass filter has a quick response and successfully removes the unwanted oscillations. The most common loop filter used is called an active PI filter (PI = proportional + integral, taken from control theory and named due to the fact that it has a pole at s = 0, hence acts as an integrator) [1]. Taken from the corresponding RC circuit filters and implemented in software using their transfer functions, the filter has the Bode plot depicted in Figure 4 [1]. The transfer function for the PI filter is given by

$$F(s) = \frac{1 + s\tau_2}{s\tau_1} \tag{2.5}$$

where in the analog circuit world, τ_1 and τ_2 are RC time constants of the circuit filter and are determined by the loop bandwidth of the system, which is in turn a function of the noise.

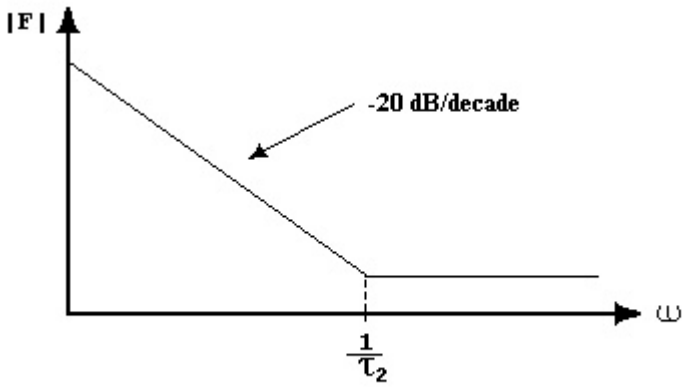

Figure 4. Bode Plot of Active PI Filter. (From Ref [1]).

3. Voltage Controlled Oscillator / Numerically Controlled Oscillator

The NCO takes as its input the output from the loop filter, $u_f(t)$, which is proportional to the phase error. By using an appropriate scaling, or gain constant K_0, of the NCO, we can adjust the output, $u_2(t)$, to account for the phase error in the two signals. The NCO has a center frequency. This is the oscillator frequency without any adjustments. This value is selected to reflect approximate working frequencies of the system. The PLL will properly lock onto a signal within a percentage range of the center frequency. Call this frequency ω_0. An adjustment made on the frequency of the NCO will be an offset from ω_0. Thus the output of the NCO has a frequency,

$$\omega_2(t) = \omega_0 + K_0 u_f(t). \tag{2.6}$$

The discrete nature of the software system requires the output of the NCO to be a phase, as opposed to frequency, and thus by definition the phase is given by the integral over the frequency variation,

$$\theta_2(t) = K_0 \int u_f(t)dt. \tag{2.7}$$

Using a look-up table, the output of the system is taken as

$$\begin{aligned} u_2(t)_{InPhase} &= \cos(\theta_2(t)) \\ u_2(t)_{QuadraturePhase} &= \sin(\theta_2(t)) \end{aligned} \tag{2.8}$$

Either the in-phase or the quadrature-phase can be used as the oscillator, but with the implementation described above, both are needed as inputs to the PD.

B. OPERATION OF THE PHASE LOCK LOOP

To understand the operation of the PLL and how the blocks work together, assume first that the system is initially locked, i.e., $\omega_1 = \omega_2$, and examine three types of signals at the input: a phase step, a frequency step, and a frequency ramp. Taken from [1], the signals are shown in Figure 5.

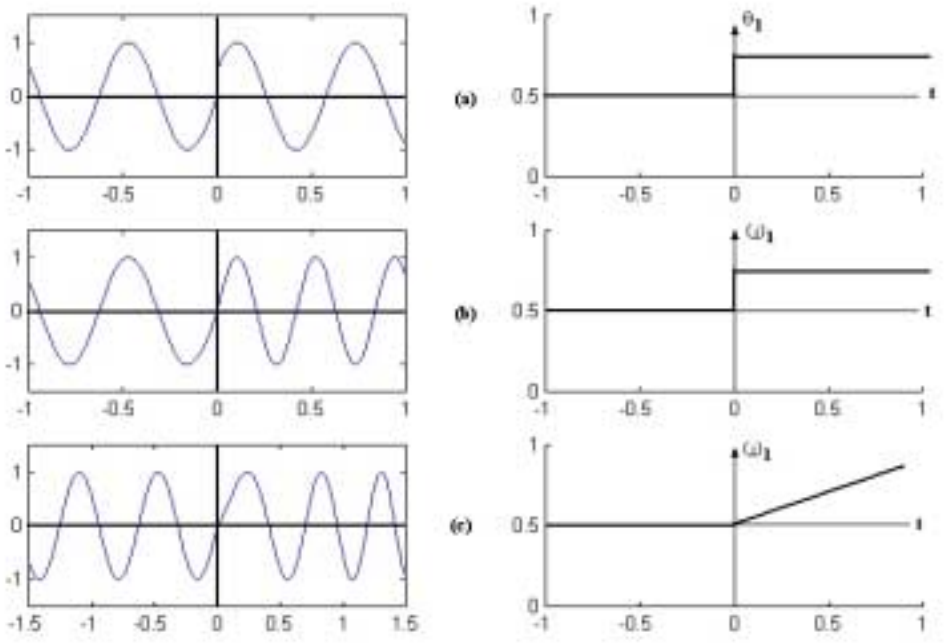

Figure 5. Exciting Functions as Applied to Input of a PLL. (a) Phase Step. (b) Frequency Step. (c) Frequency Ramp. (From Ref [1]).

The aim is to characterize these signals in terms of their phase, and proceed with analysis using a generic phase term in the signal. We assume an input signal of a sinusoid as previously done, so that $u_1(t) = U_{10}\sin(\omega_1 t + \theta_1(t))$, where $\theta_1(t)$ is a generic phase signal that represents the various excitation input signals.

For the phase step, which represents a phase-modulated signal, $\theta_1(t)$, simply performs a step change at $t = 0$, and is given by,

$$\theta_1(t) = \Delta\Phi u(t) \tag{2.9}$$

where $0 \le \Delta\Phi \le 2\pi$ is the magnitude of the phase change and $u(t)$ is the step function.

The next case is an example of a frequency-modulated signal. This signal has a step change in frequency at $t = 0$, which is given by the increment $\Delta\omega$. Thus the corresponding input signal is given by

$$\begin{aligned} u_1(t) &= \sin((\omega_1 + \Delta\omega tu(t))) \\ &= \sin(\omega_1 t + \Delta\omega tu(t)). \end{aligned}$$

Thus

$$\theta_1(t) = \Delta\omega t. \tag{2.10}$$

This phase signal is just a ramp function for $t \ge 0$.

The final signal is one whose frequency increases linearly with time. As shown in Figure 5, the frequency is a ramp function, and the rate of increase in frequency is the slope of the line. If the slope of the ramp function for frequency is $\Delta\dot{\omega}$, then the total frequency of $u_1(t)$ is $\omega_1 + \Delta\dot{\omega}t$. By definition, the frequency of a signal is the first derivative of its phase with respect to time,

$$\omega_1 + \Delta\dot{\omega} = \frac{d\theta_1}{dt}.$$

Thus the phase of the signal at time t is the integral of its angular frequency over the time interval $0 \le \tau \le t$, and so the input signal $u_1(t)$ can be written as

$$\begin{aligned} u_1(t) &= U_1 \sin \int_0^t (\omega_1 + \Delta\dot{\omega}\tau) d\tau \\ &= U_1 \sin\left(\omega_1 t + \Delta\dot{\omega}\frac{t^2}{2} \right). \end{aligned}$$

Consequently, the phase signal is given by

$$\theta_1(t) = \Delta\dot{\omega}\frac{t^2}{2} \tag{2.11}$$

1. The Transfer Function[1]

It is often helpful in examining the operation of any system to derive and understand the transfer function, $H(s)$, of the system. This will be the approach in understanding the dynamic operation of the PLL. The transfer function is defined as the Laplace transform of the output divided by the Laplace transform of the input. For a PLL, we want to relate the phase signals of the output and input, thus

$$H(s) = \frac{\Theta_2(s)}{\Theta_1(s)} \tag{2.12}$$

where $\Theta_i(s)$ is the Laplace transform of the phase signal $\theta_i(t)$. To build this mathematical model of the system, assume initially the PLL is locked, so that $\omega_1 = \omega_2$. This being the case, the output of the PD is the phase error $\theta_e = \theta_1$ times the PD gain constant K_d, i.e.,

$$u_d(t) = K_d \theta_e \tag{2.13}$$

which implies

$$H_{PD}(s) = K_d. \tag{2.14}[2]$$

The next block in the PLL is the loop filter. The transfer function of the PI active filter was already given in Equation (2.5). The last block to derive the transfer function is the VCO/NCO. Recall that the NCO adjusted the output frequency of $u_2(t)$,depending on its input from the loop filter. A negative input reduced the output frequency and a positive input increased it. The center frequency of the NCO, ω_0, was the starting point for this adjustment. The output frequency, $\omega_2(t)$, was defined by

$$\omega_2(t) = \omega_0 + K_0 u_f(t) \tag{2.15}$$

where $K_0 u_f(t)$ is the variation in the frequency. However we want the output phase of the signal, not its angular frequency. Using the definition of the phase as the integral over the frequency variation,

---

1 Transfer function analysis and derivation done using [1].

2 Note here we assume over a small time period, the phase error does not change dramatically and is thus approximated to be a constant.

$$\theta_2(t) = K_0 \int u_f(t)dt, \tag{2.16}$$

The Laplace transform of an integral is 1/s, thus

$$\Theta_2(s) = \frac{K_0}{s} U_f(s) \quad \Rightarrow \quad H_{NCO}(s) = \frac{K_0}{s}. \tag{2.17}$$

The closed-loop block diagram of the transfer function is shown in Figure 6.

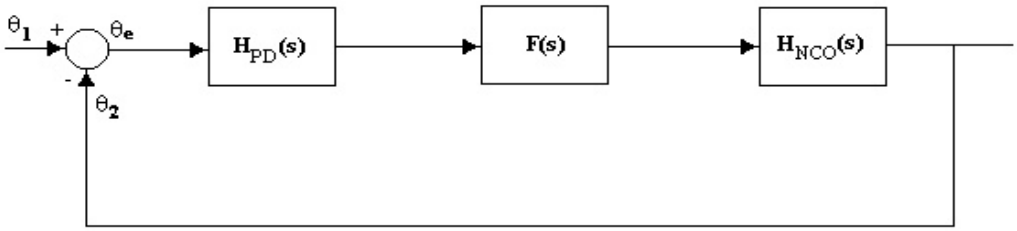

Figure 6. Block Diagram of PLL Transfer Function

From control theory, the closed loop transfer function for the entire system is given by

$$H(s) = \frac{H_{PD}(s)F(s)H_{NCO}(s)}{1 + H_{PD}(s)F(s)H_{NCO}(s)} = \frac{K_d K_0 F(s)}{s + K_d K_0 F(s)}. \tag{2.18}$$

Simplifying this to standard notation, Equation (2.18) becomes

$$H(s) = \frac{\frac{K_d K_0}{\tau_1}(1 + s\tau_2)}{s^2 + s\left(\frac{K_d K_0 \tau_2}{\tau_1}\right) + \frac{K_d K_0}{\tau_2}}. \tag{2.19}$$

Control theorists put this in terms of natural frequency, ω_n and damping factor, ζ, in order to use a standard equation which can easily be understood and studied. The equivalent transfer function equation in *normalized form* is given by

$$H(s) = \frac{2s\zeta\omega_n + \omega_n^2}{s^2 + 2s\zeta\omega_n + \omega_n^2}, \tag{2.20}$$

where

$$\omega_n = \sqrt{\frac{K_0 K_d}{\tau_1}},$$

and

$$\zeta = \frac{\omega_n \tau_2}{2}.$$

To calculate the time response of the transfer function, the inverse Laplace transform of (2.20) is required. The time domain equivalent equation becomes

$$h(t) = 1 + \frac{1}{2(\zeta^2 - 1)} e^{-\omega_n \zeta t} \left(2(1-\zeta^2)\cos\left(\omega_n \sqrt{1-\zeta^2}\right) + 2\zeta\sqrt{1-\zeta^2}\sin\left(\omega_n \sqrt{1-\zeta^2}\right)\right) \qquad (2.21)$$

which is simply a damped oscillation.

To get a feel for the dynamic operation of the PLL for the three Reference inputs described above, we look at the transient response of the transfer function in normalized form for different values of ω_n and ζ. Then finding a desired response, values to build the PLL can be determined.

a. Phase Step Applied to Input

To see a typical transient response, a phase step was applied to the input at $t = 0$, corresponding to the first type of input above, so that

$$\theta_1(t) = \Delta\Phi u(t).$$

Figure 7 shows the error time response of the transfer function versus time for various values of ζ.[3] A natural frequency of 200 rad/sec is used. The value of ω_n only scales the x-axis, not the

[3] The time response can be obtained by taking the inverse Laplace transform of the transfer function. For this transfer function the time response is given by Equation (2.21)

response of the system or its characteristics. To note is the fact that as $t \to \infty$, the error approached zero. This can also be derived from the transfer function by finding the error transfer function, defined by

$$H_e(s) = \frac{\Theta_e(s)}{\Theta_1(s)} = \frac{s}{s + K_d K_0 F(s)} \tag{2.22}$$

and using the final value theorem of the Laplace transform which states

$$\theta_e(\infty) = \lim_{s \to 0} s\Theta_e(s) = 0 .$$

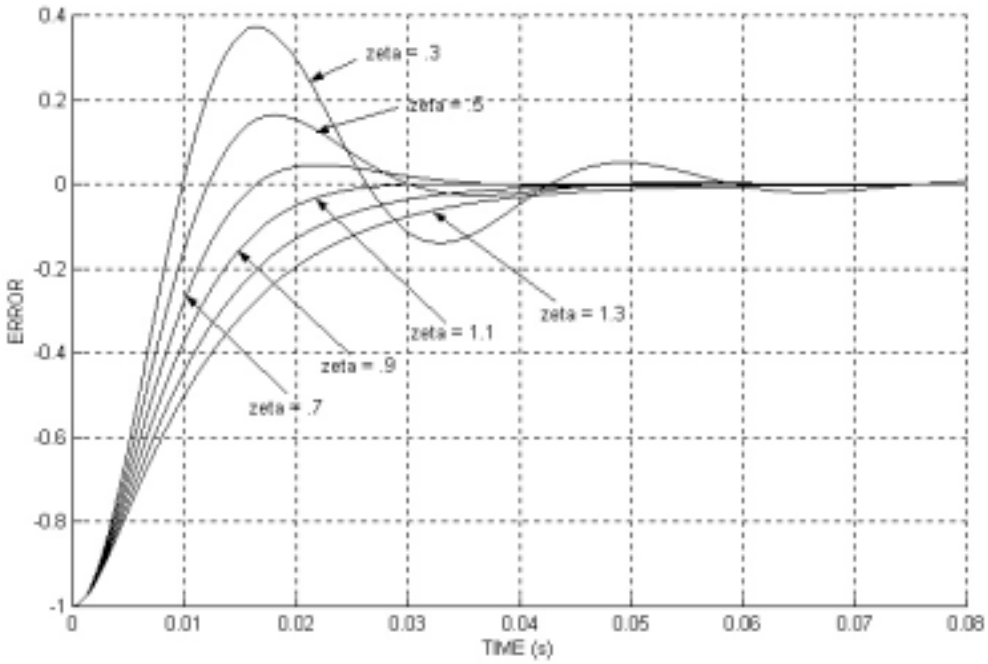

Figure 7. Time Response of PLL for different values of ζ and $\omega_n = 200$.

b. Frequency Step Applied to Input.

Again applying the final value theorem, a frequency step corresponds to a transfer function input of

$$\Theta_1(s) = \Delta\omega / s$$

Performing the same calculation on the phase error transfer function,

$$\theta_e(\infty) = \lim_{s\to 0} \frac{\Delta\omega}{s^2} \frac{s^2}{s^2 + 2s\zeta\omega_n + \omega_n^2} = \frac{\Delta\omega}{\omega_n^2} \tag{2.23}$$

This value approaches zero for small frequency steps and high gain loops, which increases the value of ω_n

c. Frequency Ramp Applied to Input

Using similar techniques for a frequency ramp, the results are less encouraging. The error is $\theta_e(\infty) = \Delta\dot{\omega}/\omega_n^2$, where $\Delta\dot{\omega}$ is the rate of change of the input signal frequency. Thus for a large rate of change in frequency, the PLL unlocks. Experimen-tation with PLL's has shown that a practical design limit to the rate of change in frequency is,

$$\Delta\dot{\omega}_{MAX} = \frac{\omega_n^2}{2}. \tag{2.24}$$

C. PERFORMANCE MEASURES OF THE PLL

For a full derivation of the following key PLL parameters, one is encouraged to see Reference [1]. For the purposes of this research, it is sufficient to define the parameters and state the results for a given PLL, and shed some light on the competing parameters and their interaction. The focus will be on five key parameters that govern the dynamic performance of the PLL

- The lock range, $\Delta\omega_L$
- The lock time, T_L
- The pull-in range, $\Delta\omega_P$
- The pull-in time, T_P
- The pull-out range $\Delta\omega_{PO}$.

Additionally, a figure of merit for the PLL of loop bandwidth, B_L will be given consideration during the study of PLL performance in the presence of noise.

1. The Lock Range

The lock range is defined as the range of frequency offset between the center frequency of the VCO and the reference frequency in which the PLL locks within one single-beat note between reference and output frequencies. We assume that the PLL is initially unlocked and switched on at time $t = 0$. To derive this figure of merit, we assume a simple PD of multiplying the input reference signal and the NCO output signal to produce the phase error. If this is the case, then for a frequency offset of $\Delta\omega$,

$$u_d(t) = K_d \sin(\Delta\omega t) + \text{higher-frequency terms} \tag{2.25}$$

where the higher-frequency terms are discarded due to the loop filter. Looking at the output of the loop filter, the result,

$$u_f(t) = K_d \,|\, F(\Delta\omega) \,|\, \sin(\Delta\omega) \tag{2.26}$$

is obtained. The output of the LF, $u_f(t)$, is just an ac signal causing a frequency modulation of the NCO, with a peak frequency deviation of $K_0 K_d \,|\, F(\Delta\omega) \,|$. For lock to occur, this peak frequency deviation must be less than the frequency offset $\Delta\omega$. If the offset is larger than this frequency deviation, lock cannot occur in a single cycle. The lock range can be determined by determining when this peak frequency deviation is just as large as the frequency offset. Thus the equation for locking becomes

$$\Delta\omega_L = K_0 K_d \,|\, F(\Delta\omega_L) \,|\,. \tag{2.27}$$

This is a non-linear equation, but to solve it a practical approximation can be made. From the Bode plot of the loop filter transfer function in Figure 4, the lock range is greater than the cut-off frequencies $1/\tau_1$ and $1/\tau_2$. Thus a conservative approximation is $|\, F(\Delta\omega_L) \,| = \tau_2 / \tau_1$. Using Equation (2.20), the lock range becomes

$$\Delta\omega_L \approx 2\zeta\omega_n\,. \tag{2.28}$$

Simulations and experiments have shown that this is a conservative approximation and can be used confidently in the design process. Clearly a larger lock range is desirable.

1. Lock Time

Lock time is the time the PLL takes to lock onto a signal when it is initially unlocked, given that the reference signal is within the lock range. When $\zeta < 1$, the transients of the damped oscillation die out after one cycle, thus the lock time can be approximated accurately as

$$T_L \approx \frac{2\pi}{\omega_n}. \tag{2.29}$$

2. The Pull-in Range

The lock range is a subset of the pull-in range, which is defined as the range of frequency offset in which the PLL will eventually lock after a number of cycles. This is dependant on the type of loop filter used. For the active PI filter, the pull-in range is infinite, i.e.,

$$\Delta\omega_p \to \infty, \tag{2.30}$$

thus any reference frequency input to the PLL will eventually be locked onto. For different types of loop filters, with a finite gain at dc, the pull-in range is decreased. See [1] for further details on pull-in range for different filter types.

3. Pull-in Time

Pull-in time is the time required for a PLL to lock onto a frequency within the pull-in range. Defining $\Delta\omega_0$ as the frequency offset $\omega_1 - \omega_2$, the pull-in time is determined to be

$$T_P = \frac{\pi^2}{16}\frac{\Delta\omega_0^2}{\zeta\omega_n^3} \tag{2.31}$$

Note that, as $\Delta\omega_0 \to \infty$, the pull-in time approaches infinity. The larger the offset, the longer the pull-in time.

4. The Pull-out Range

The pull-out range is defined as that frequency step which causes the PLL to unlock if applied to the reference input. Direct calculations of the pull-out range are not possible, but in [5], computer simulations have arrived at the following approximation

$$\Delta\omega_{PO} = 1.8\omega_n(\zeta + 1). \qquad (2.32)$$

The relationship of the frequency ranges is depicted in Figure 8. Starting from the center frequency of the system, the lock-range is the smallest of the range parameters. Next is the pull-out range, and finally the pull-in range has the largest. The range of values outside the pull-in range is called dynamically unstable due to the fact when operating dynamically, the system will not lock if a reference frequency in that range is encountered. Inside the pull-in range, and outside the lock range is called conditionally stable. This means the system will eventually lock on frequencies in this range, but could unlock temporarily with a reference frequency in this range.

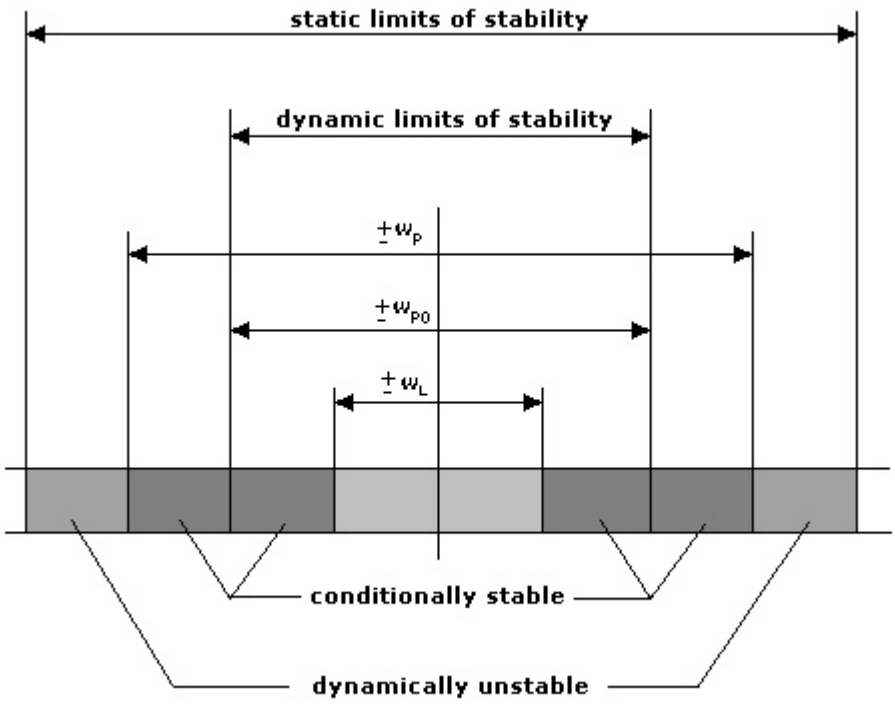

Figure 8. Relationship of frequency ranges of a PLL. (From Ref. [1].)

D. PERFORMANCE OF THE PLL IN NOISE

An in-depth analysis of the performance of the PLL in noise is outside the scope of this research; however, some insight into what can be expected from a PLL operating in the real world is required. The following definitions are used and assumptions made in this discussion of noise.

1. $SNR_i = 10\log\left(P_S/P_n\right)$, where P_S is the reference signal power and P_n is the noise power, is the signal-to-noise ratio at the input of the PLL.

2. All noise is assumed to be Additive White Gaussian Noise (AWGN), meaning that it has a flat power spectral density.

3. A low-pass pre-filter of bandwidth B_i is implemented prior to the PLL.

4. The *noise bandwidth* is defined as $B_L = \int_0^\infty |H(j\omega)|^2 \, d\omega$, where $H(j\omega)$ is the transfer function of the PLL. Inserting equation (2.20) for $H(j\omega)$, the integral solves to

$$B_L = \frac{\omega_n}{2}\left(\zeta + \frac{1}{4\zeta}\right). \qquad (2.33).$$

5. The signal-to-noise ratio at the output is defined by

$$SNR_L = SNR_i \frac{B_i}{2B_L}. \qquad (2.34)^4$$

This figure is helpful in determining how often the PLL unlocks due to noise, and the SNR_i required to ensure a lock.

Recall that the lock range, $\Delta\omega_L$, from Equation (2.28) is proportional to the natural frequency and is desired to be as large as possible; hence, theoretically, designing the PLL with a ω_n as large as required is an easy task, but with the introduction of noise, and Equation (2.33), the larger ω_n, the larger is B_L. This suggests a trade-off between reducing the noise sufficiently and a larger lock range, which is exactly the case. To increase the lock range of the system, a larger SNR must be used. If only a small SNR is available, B_L must be made small, which reduces $\Delta\omega_L$.

[4] For a derivation of SNR_i, one is directed to Ref [1].

The same is true of the lock time. This parameter is inversely proportional to ω_n, and thus larger values of ω_n reduce T_L. Again, by reducing T_L, B_L is increased.

Practical experiments with PLLs have shown that for stable operation, we need an $SNR_L \geq 6$ dB [1]. A system could eventually lock with a lower SNR_L, but the phase jitter would cause frequent unlocks and the system would be of little value. This leads to the final important figure, that of how often a PLL system, on average, will unlock. T_{av} is defined as the average time interval between two unlocks of the system. Figure 9 is taken from [2], which depicts T_{av} as a function of SNR_L. For high SNR_L, T_{av} is very difficult to find or measure, thus Figure 9 is provided to give an idea of the relationship.

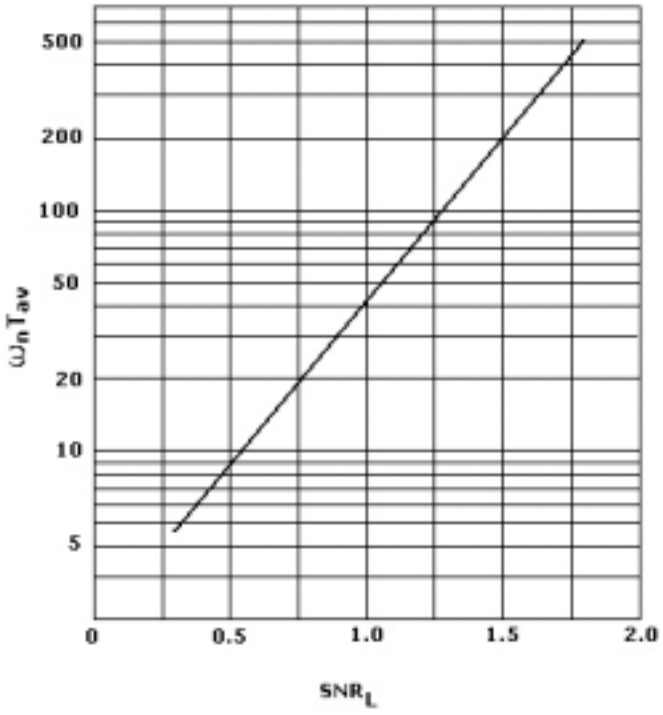

Figure 9. T_{av} plotted as a function of SNR_L, where T_{av} is normalized to the natural frequency. (From Ref. [2].)

A phase lock loop is a feedback system designed to lock onto the frequency and phase of the input reference signal. This is done using a phase detector, a loop filter, and a numerically

controlled oscillator. This chapter described the operation of these components and derived a transfer function for the system. Using this transfer function a typical time response for the system was plotted. Important figures of merit were defined and derived for the PLL. These performance measures are lock range, lock time, pull-in range, pull-in time and pull-out range. Finally a section describing the operation of the PLL in a noisy environment and how a noise can affect the figures of merit of a system was included.

The objectives of this research are to implement the phase lock loop using fixed-point arithmetic. In order to do this a study of fixed-point arithmetic is needed. The following chapter introduces fixed-point arithmetic and some of the considerations that need to be looked at when using this type of representation.

THIS PAGE INTENTIONALLY LEFT BLANK

III. FIXED-POINT ARITHMETIC

Real world values are approximated to nearly any degree of error in computers using floating-point arithmetic. While the precision afforded by floating point arithmetic is advantageous in numerous applications, the speed at which a computer can calculate and manipulate floating-point numbers is a disadvantage. Binary fixed-point arithmetic takes advantage of the processor shift instruction when multiplying or dividing by two to speed up arithmetic, simply by representing numbers in a different fashion. The second advantage to fixed-point arithmetic is storage space. If the range of the real world values are known, and does not need 32-bits to cover the range adequately, fixed-point numbers can be scaled to cover the range and use perhaps half number of bits. To gain an understanding of how fixed-point arithmetic works, an introduction to floating-point representations is given.

A. FLOATING-POINT NUMBERS

The IEEE standard 754 has dominated most of today's processors for floating point arithmetic. It specifies four formats, the two most common being single-precision and double-precision. Single precision uses 32-bits and double precision uses 64-bits. Both double and single precision formats contain three components: a sign bit (S), a fraction field (f), and an exponent field (e). Figure 10 shows how the bits are allocated for IEEE standard 754 floating-point format for both single and double precision. As the name implies, the fraction field is fixed and the radix point, defined by the exponent field as described below, is variable. Subsequently, for small integer number, most of the f field bits are zeros, hence wasted space.

The exponent field is expressed as 2^{e-127} for single precision and 2^{e-1023} for double precision where e is a variable between 0 and 256. The field scales the fraction field and places the radix point accordingly. The range of numbers able to be expressed

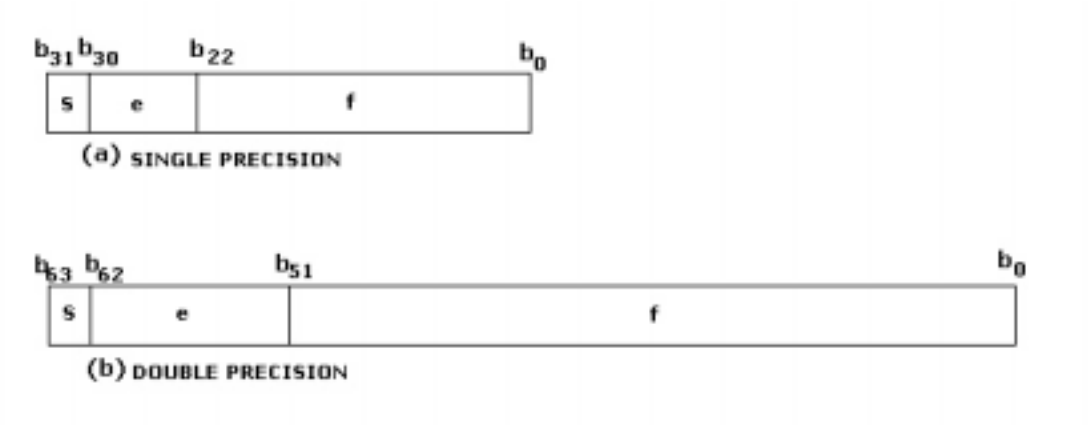

Figure 10. IEEE 754 Floating Point Number Format

is a function of the exponent field, which ranges from $2^{-126} \approx 10^{-38}$ ($e = 0$) to $2^{128} \approx 10^{38}$ ($e = 256$)[5].

The fraction field, as already alluded to, is fixed, and therefore is responsible for the precision of the number. The precision is defined as the distance between two subsequent representable numbers. This is simply 2^{-f}, where f is the number of bits in the f-field. For single precision floating point, the precision is $2^{-23} \approx 10^{-7}$.

B. FIXED-POINT NUMBERS

As can probably be concluded, fixed-point numbers fix the radix point and the fraction field is the variable. This has two distinct advantages: 1.) by allowing the fraction to be variable in size, the storage space for an integer can be tailored to the range of values need, and 2.) arithmetic is essentially integer arithmetic and, when done in binary, this amounts to shifts, rather than cumbersome multiplications.

1. Fixed-point Number Representation

Fixed-point numbers can be specialized to unsigned and signed integers, or fractionals; however, the most general form will be considered here, which is a signed general fixed-point number, either an integer, fraction, or combination. To understand the usefulness of fixed-point numbers, a more intuitive look at decimal fixed-point arithmetic will be presented, followed by

[5] A bit is needed for the possibility of exceptional numbers such as infinity or NaN.

the extension to full binary fixed-point arithmetic. Fixed-point numbers in a processor are represented solely as integers. In order to properly insert the decimal point, a processor has a syntax for all numbers. For instance, if the processor uses a S.3.2 syntax, this means a sign bit (S), 3 integer digits, and 2 fractional digits (realize that here a 16-bit range is being used to specify the sign bit and the five decimal digits). The number 238.15 in fixed-point S.3.2 notation is represented as the integer 23815. The original number is multiplied by 10^2, where the exponent two is taken from the third field. To add two fixed-point numbers, conversion to the notation of the number with the largest third field is required. A numerical example will help to illuminate the process. Suppose we have the number above, 238.15, in S.3.2 notation and want to add the number 145.6 in S.4.1 notation. The processor has the integer numbers 23815 and 1456 stored, but to add the two, it must convert both numbers to S.3.2. The whole process is as follows

$$238.15 * \frac{100}{100} = \frac{23815}{100} \quad \text{and} \quad 145.6 * \frac{10}{10} = \frac{1456}{10}$$

$$\frac{23815}{100} + \frac{1456}{10} = \frac{23815}{100} + \frac{1456}{10} * \frac{10}{10} = \frac{38365}{100}$$

which is 383.65 in S.3.2 format.

The payoff comes when we extend this to binary fixed-point numbers. Instead of multiplying by powers of 10, we multiply by powers of 2. In a processor, this is accomplished with shifts. Shifts in a processor are extremely fast, thus speeding up computationally heavy applications.

Of course a processor does not keep track of the format as described above, but the actions are equivalent. A fixed-point number is represented by

$$V = SQ + B \tag{3.1}$$

where V is the real world number, B is the bias, which is used for signed numbers, Q is the integer that encodes V, and $S = 2^E$ is the slope or scaling of the number. The scaling determines where the radix point belongs.

2. Precision and Range in Fixed-point Arithmetic

The range of a number gives the upper and lower limits of representation, while the precision or resolution gives the distance between two successive numbers in the representation. In fixed-point numbers these are a function of the scaling, *S*, and the number of bits used or length of *V*. The range and precision of a number are in constant competition. For a large range the resolution must be reduced. Conversely, for a high resolution the range is decreased. To understand this relationship, calculations of both of important parameters will be made. The terminology that is used will be a high or large resolution is desirable, which is equivalent to a large or high precision. Often the term large precision indicates that the distance between two successive representable numbers is large. For the purposes of this research, a large precision indicates a high degree of accuracy, and a low or small precision indicates a low degree of accuracy.

Precision is determined by the least significant bit (LSB), more specifically, what effect changing the LSB from a zero to a one has on the number. Changing the LSB is the smallest increment that can be made, thus the effect of changing the LSB determines the precision. So what does the LSB represent? The scaling factor, *S*, determines where the radix point belongs, hence, determines what the LSB represents. Suppose $E = 0$, i.e., $S = 2^0$. This implies the radix point is to the right of the LSB, and thus a change from a zero to a one in the LSB results in a change in the number of 1. The resolution of this scaling is 1. For a better resolution, E is required to be negative. For $E = -8$, the resolution is $2^{-8} = 0.00390625$. Generally, the precision is defined as the scaling $S = 2^E$.

This increased precision, however, comes at a price. Assume we are using *n* bits to represent the fixed-point numbers. This means that a total of 2^n possible numbers can be represented. It should be clear that if a finite number of values can be represented, the greater the precision, the lesser the range. To determine the range, simply multiply the precision by 2^n which gives the total range of numbers available. Recall the bias, *B*, is simply a sliding window for the range of numbers. Hardware represents numbers from 0 to 2^n, thus for signed numbers a negative bias, $B = -2^{n-1}$ is used, which slides the range between -2^{n-1} and 2^{n-1}. Table 1 shows the range and precision of a 16-bit signed and unsigned fixed-point numbers for various scalings using a zero bias for the unsigned case and $B = 2^{n-1}$ bias for the signed.

Scaling	Precision	Range of Signed Values (low,high)	Range of Unsigned Values (low,high)
2^{-3}	0.125	-4096,4095.875	0,8191.875
2^{-4}	0.0625	-2048,2047.9375	0,4095.9375
2^{-5}	0.03125	-1024,1023.96875	0,2047.96875
2^{-6}	0.015625	-512,511.984375	0,1023.984375
2^{-7}	0.0078125	-256,255.9921875	0,511.9921875
2^{-8}	0.00390625	-128,127.99609375	0,255.99609375
2^{-10}	0.00097656	-32,31.999023	0,63.999023
2^{-12}	0.000244140	-8,7.9997559	0,15.9997559
2^{-15}	0.000030518	-1,0.99996948	0,1.99996948

Table 1. Range and Precision of a 16-bit Fixed-point Data Type

3. Errors in Fixed-point Numbers

Due to the relationship between range and precision in fixed-point arithmetic, errors occur between real world values and fixed-point representations of those values. Therefore, it is important to know the ranges and values of the required numbers and how important accuracy is when using fixed-point arithmetic. Errors can occur in essentially two ways, the value being represented is outside the range of the scaling used and error due to a too low resolution.

The first kind of error is often easier to find. In an application, an out-of-range error, while not necessarily raising an error message flag, can be found through analysis. Most fixed-point processors, when encountering a value outside the fixed-point range, saturate the fixed-point representation to the max or min of the range. This can be a very bad approximation of the real world value. Thus knowing the range of values in an application is vital so that proper scaling can be used.

A low resolution can also lead to errors which may be difficult to find. A processor representing a number between two successive fixed-point numbers does one of three things: it rounds down, rounds up, or rounds to the nearest representable value. To reduce errors as much as possible, rounding to the nearest representable value is preferred. This reduces the error to half the precision and was the approach used for the PLL simulation in Chapter IV. Care must be taken when arithmetic is done in fixed-point because adding, subtracting and multiplying can have dramatic effects on the size of numbers. Subtracting two similar numbers can result in a number smaller than the precision, hence equates to zero. Equally troubling is adding a small number to a large number. The larger number scaling must be used because the larger range is required, but this means low precision. If the small added number is smaller than the precision of the larger number, the addition will have no effect. It will be equivalent to adding zero.

While increasing the number of bits can increase precision and range to any amount, this is not a valid solution for practicality reasons. In today's commercial market, 32-bit fixed-point processors are the largest available. Application-specific DSPs are made with a larger number of bits, but these are not readily available nor cheap. Careful consideration of application values and ranges must therefore be a priority when using fixed-point mathematics.

Using a fixed-point arithmetic implementation requires knowledge of the range of values and precision required for the application. The range and precision is determined by the scaling used. For the fixed-point PLL in the next chapter, each input and output range was determined from the floating-point model, then appropriate scaling used at each level. The next chapter details the procedure in developing the floating-point model and conversion to a fixed-point simulation.

IV. FIXED-POINT PLL SIMULATION

The PLL simulation developed in this research used Mathwork's Simulink software package. The reasons for choosing Simulink for this application are numerous. This software is designed for control systems and has an imbedded fixed-point blockset. This feature negated the requirement to generate code to simulate fixed-point arithmetic. Another desirable feature of Simulink is the Real Time Workshop, which generates C code from a simulation model. This feature will allow easy implementation on an FPGA as C can be easily converted to VHDL, the language required for programming an FPGA. Finally, the Simulink software was available and familiar. There was no need to buy expensive modeling or simulation software, and because the software was familiar, learning a whole new language and set of procedures was negated.

A. PROCEDURE FOR MODEL

The procedure for building the model was first to construct a floating-point simulation in MATLAB using the design from Reference [1]. Having a working floating-point model gives expected results of a PLL simulation and a comparison for the fixed-point model. The next step was to implement the floating-point model in Simulink, the block diagram based simulation software by Mathworks. This done, the Simulink model was converted to a fixed-point version where analysis can begin.

1. Floating-point MATLAB Simulation

Any software simulation that takes a signal as input must rely on sampling that signal and using the discrete data points. Thus to the software, a vector of data of a 20 hertz signal looks no different than a vector of data bits of a 200 kHz signal. As long as the sampling frequency is known and is larger than the Nyquist limit, the actual frequency of the input signal can be arbitrary. The vector representing the input signal is a vector of numbers, regardless of the actual frequency of the analog signal it approximates.

a. Input signal Assumptions

For the purposes of the simulation the center frequency, F_0, of the NCO was taken to be 2 kHz. This just means the expected frequency of the input signal is "around" 2 kHz. A sampling frequency F_s, of 40 kHz was used. While a sampling frequency this high is not required for a 2 kHz signal, the resulting output is much easier to analyze visually. The input and output amplitudes of the signals were unity, with no DC offset. This assumption is justified by realizing that the input signal can simply be normalized by its amplitude if other than one, and any DC offset can be removed by filter prior to input into the PLL. A Hilbert transform is performed on the real input signal creating a complex signal with in-phase and quadrature-phase components. The signal was initially given a random frequency F_C, within 2.5% of the center frequency of the NCO. This assured the input signal to be within the lock range. It was also given a random phase between 0 and π. The signal to noise ratio (SNR) at the input to the PLL was assumed to be $SNR_i = 15$ dB. This noise level assured a lock was possible, but also gave an idea of how noise affected the system. A pre-filter B_i, of 1 kHz around the expected input signal of 2 kHz was assumed. This would leave the input signal around 2 kHz undistorted, but reduce the effects of noise. Summarizing the input signal assumptions,

- $F_0 = 2$ kHz.
- $F_S = 20$ kHz.
- $B_i = 1$ kHz.
- $u_1(t) = \cos(2\pi F_C t + po) + i\sin(2\pi F_C t + po) + n(t)$.

where $F_C \in [1950, 2050]$, $po \in [0, \pi]$, and $n(t)$ is the noise with $SNR_i = 15$ dB.

b. Determination of PLL Parameters

The first step in developing an algorithm for a software PLL is to determine appropriate parameters depending on the application and situation. Typical communications signals have an SNR of 15 dB, thus we are assured that the PLL will lock, although this will be verified once all parameters are determined. With the noise bandwidth, B_L, not a factor, determining the lock range $\Delta\omega_L$ is required. If the expected input signal has a frequency of

around 2 kHz, a fair estimate is that the actual frequency of the signal will not vary by more than 5%. This means we need a lock range of 100 Hz, or $\Delta\omega_L \geq 200\pi$ rad/s. Setting $\Delta\omega_L = 200\pi$ rad/s and using equation (2.28), the natural frequency in terms of the damping factor is easily determined.

$$\omega_n = \frac{\Delta\omega_L}{2\zeta} \text{ rad/s}. \tag{4.1}$$

From control theory, the transfer function of the system is optimally flat for $\zeta = 0.707$, but from an analysis of noise, the noise bandwidth as a function of ζ is flattest for $\zeta = 0.5$, thus a compromise of $\zeta = 0.6$. With this information the resulting natural frequency is $\omega_n = 166.66\pi$ rad/s.

Both PD gain and loop filter gain were set equal to one. The final parameters to be determined were the lead and lag constants of the loop filter. The transfer function of the loop filter was given in Equation (2.5). To determine the constants, the transfer function needs to be related to the time domain input-output. To do this, the transfer function of Equation (2.5) is converted to its equivalent z-transform. This is done using the bilinear transform, such that,

$$F(z) = \frac{b_0 + b_1 z^{-1}}{1 + a_1 z^{-1}} \tag{4.2}[6]$$

where,

$$a_1 = -1$$

$$b_0 = \frac{T_S}{2\tau_1}\left(1 + \frac{1}{\tan\left(\frac{T_S}{2\tau_2}\right)}\right)$$

[6] For a complete discussion on bilinear transforms see [4].

$$b_1 = \frac{T_S}{2\tau_1}\left(1 - \frac{1}{\tan\left(\frac{T_S}{2\tau_2}\right)}\right)$$

and $T_S = 1/F_S$.

The inverse of Equation (4.2) must be found in order to derive the corresponding difference equation which will enable the loop filter to be implemented in software. Recall the transfer function in the z-domain, $F(z)$, is the input divided by the output. From Figure 3, the input to the loop filter in the sampled digital world is $u_d(n)$ and the output is $u_f(n)$. Equation (4.2) can be written

$$U_f(z) = U_d(z)F(z). \tag{4.3}$$

Substituting in $F(z)$,

$$U_f(z) = U_d(z)\frac{b_0 + b_1 z^{-1}}{1 + a_1 z^{-1}} \Rightarrow U_f(z)(1 + a_1 z^{-1}) = U_d(z)(b_0 + b_1 z^{-1}), \tag{4.4}$$

and taking the inverse z-transform back to the time domain, the time difference equation becomes,

$$u_f(n) = -a_1 u_f(n-1) + b_0 u_d(n) + b_1 u_d(n-1). \tag{4.5}$$

Equation (4.5) lends itself well to a software implementation.

The topic of noise and whether a sufficient SNR at the input signal is present was glossed over previously. Because of the high noise tolerance of PLL's, high SNR's is more of an issue in decoding and demodulating the signal than it is in locking onto the signal. For this reason SNR's for locking are exceeded to get better demodulation and bit error rate (BER) curves. However, this topic will be addressed now.

Recall that the SNR at the input of the PLL, SNR_i, is related to SNR_L, the SNR at the output of the PLL by Equation (2.34). This relation is a function of two parameters: the prefilter bandwidth and the noise bandwidth. Assuming a typical prefilter bandwidth of 1000 Hz, the only variable in the relationship is the noise bandwidth, B_L. The noise bandwidth is related to the natural frequency and the damping factor by Equation (2.33). This equation results in

$B_L = 63.5$ Hz. Equation (2.34) gives the output $\text{SNR}_L = 23.9$ dB. Experimentation has shown that PLL's will lock with $\text{SNR}_L \geq 6$ dB, thus the PLL with the above parameters should be well with the locking limits with regards to noise. The simulation parameters found are listed below for ease of reference.

- $\omega_n = 166.66\pi$ rad/s.
- $\zeta = 0.6$.
- $\Delta\omega_L = 200\pi$ rad/s (100 Hz).
- $\tau_1 = 3.64e^{-6}, \quad \tau_2 = 2.3e^{-3}$.
- $a_1 = -1, \quad b_0 = 635.1475, \quad b_1 = -621.4397$.

c. The MATLAB Simulation

With the parameters established above, the algorithm needed to be determined. The feedback loop of the PLL was done using a simple MATLAB *for-end* loop. This was done for convenience because an input signal of a given length was assumed for the simulation, thus the number of times required to "feedback" was known. This simulated the fact that real radio signal will be sent in "blocks" or "packets" in which the length of each packet is known. Inside the loop are the three blocks of the PLL.

The first block is the phase detector (PD). This block was implemented exactly the way the PD in Chapter 2 was described by using the in-phase and quadrature-phase components of the signal and some trigonometric properties. Instead of continuous signals being multiplied and manipulated, the sampled data at corresponding time intervals was manipulated. The output, $u_d(n)$, of the PD is the scalar number representing the phase error at time *n*.

The signal $u_d(n)$ was then sent to the loop filter to get rid of the higher frequency terms from the PD. The loop filter was implemented directly from Equation (4.5), producing the output signal $u_f(n)$, which is the error in the two signals at the PD.

The final block in the loop is the NCO. This block needed to take the output signal of the loop filter and make a correction to its oscillation depending on the magnitude of $u_f(n)$. From the discussion of the NCO in Chapter 2.A.3, the phase of the continuous time signal is given by Equation (2.7). In the discrete realm, the phase change of $\theta_2(t)$ can be determined by

$$\Delta\theta_2 = (\omega_0 + K_0 u_f(n))T_S . \qquad (4.6)$$

where ω_0 is the center frequency of the NCO in rad/s. With the change in phase known, the total phase of the signal can be determined by

$$\theta_2(n+1) = \theta_2(n) + (\omega_0 + K_0 u_f(n))T_S . \qquad (4.7).$$

If θ_2 is initializd to $\theta_2(0) = 0$, this computation becomes possible. Notice that Equation (4.7) is iterative and theoretically θ_2 can continue to grow, thus a simple *if-then* statement is used to bound θ_2 between $-\pi$ and π. This is done by subtracting 2π from θ_2 whenever $\theta_2 \geq \pi$. Because the output of the NCO is required to be the in-phase and quadrature-phase components, a look-up table is used to get the desired output,

$$\begin{aligned} u_2(n)_I &= \cos(\theta_2(n)) \\ u_2(n)_Q &= \sin(\theta_2(n)). \end{aligned} \qquad (4.8).$$

This signal is now sent to the PD, and the loop is complete. A complete listing of the code used for the MATLAB simulation is given in Appendix A.

While a full analysis of the performance of the above PLL is not the goal nor helpful in this research, confirming the PLL algorithm operated correctly is required. The input signal used had the assumptions described in Subsection 4.A.1.a. However, instead of a random frequency, an input signal of 2100 Hz is used. This signal represents a 5% error in the center frequency of the model and the input signal, which is the worst case expected. Figure 11 shows the results of the simulation. The output frequency of the NCO is shown in (a) and the phase error of the two signals is shown in (b). The output can be seen to oscillate, but is damped and settles on the correct frequency. The error plot can also be seen to oscillate, but dies out and goes to zero.

Evaluating lock-time, recall, $T_L = 2\pi/\omega_n = 0.0120$ s, which from the plots in Figure 11 appear to be accurate. The oscillations have died out at this time and lock has been achieved.

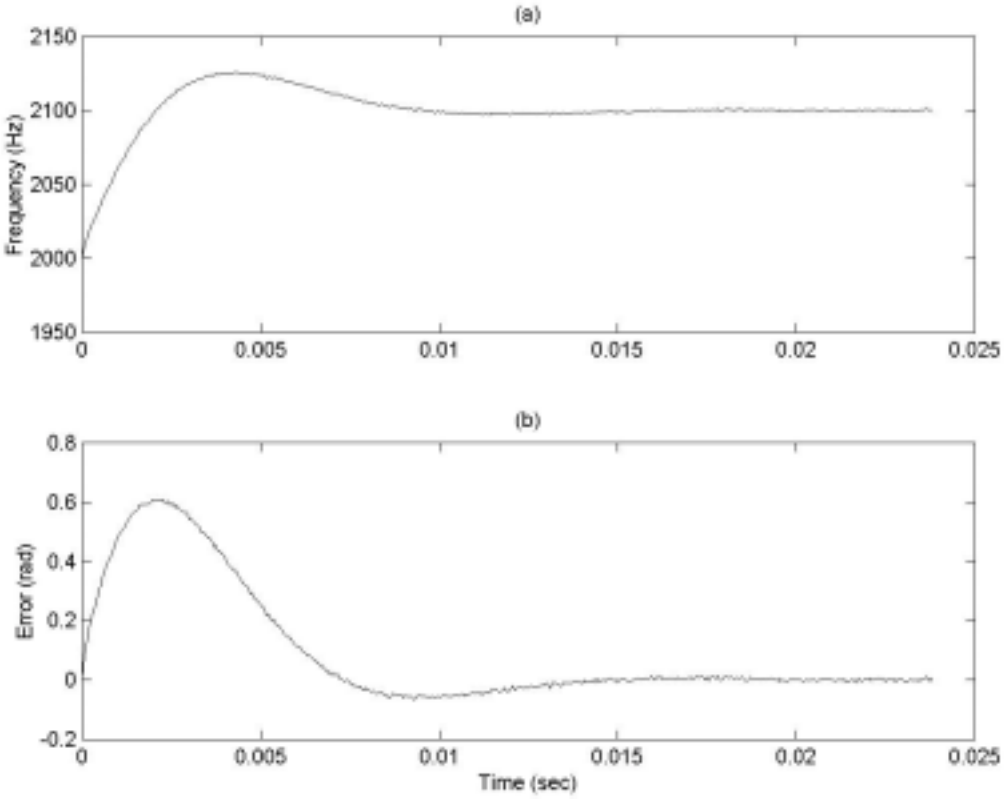

Figure 11. (a) Frequency Time Response of PLL. (b) Phase Error Time Response of PLL.

Similar evaluations on the pull-in range, which is theoretically infinite, proved to be as accurate as testable in the simulation. The pull-in time of the system is exponential with the initial frequency displacement. Proving this is the case is not plausible for all ranges of frequencies. However, given an initial frequency larger than the lock range of the system, the PLL indeed pulled the NCO output to the larger frequency offset and eventually did achieve lock. As seen in Figure 12, the output of the PD eventually reaches zero after going through several oscillations. This figure was generated with an input signal of 2350 Hz, which is an offset from the center frequency of the system of 350 Hz, clearly outside the lock range of 100 Hz. From Equation (2.31), the pull-in time of the system should be 0.0346 seconds. This is a conservative time estimate. This can be verified from the plot, where the oscillations are completely damped out.

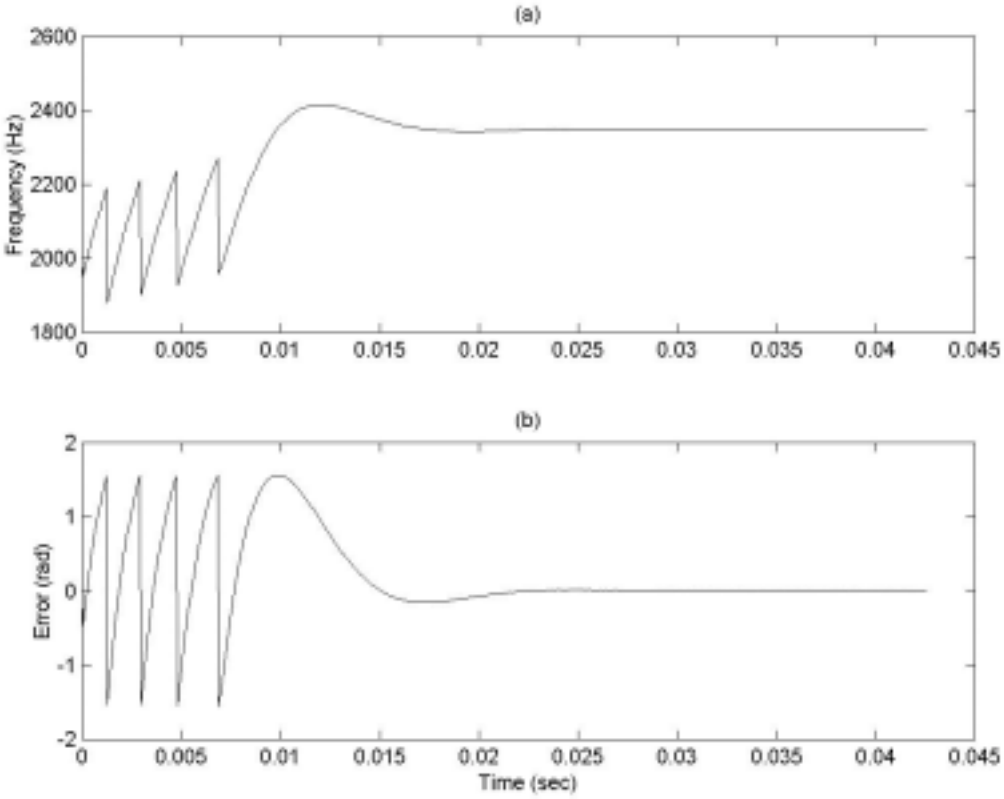

Figure 12. Time Response of PLL for an Input Frequency Larger than the Lock Range. (a). Output of System Demonstrating the Signal Locks on a Frequency of 2350 Hz. (b). Plot of the Error of the System.

This concluded the floating point MATLAB simulation. The next step was to build this system in the control simulation language of Simulink.

2. Fixed-Point Simulink Model

The fixed-point blockset in MATLAB is a toolbox used in Simulink. Simulink is a supplement to MATLAB for building models and running simulations, thus ideal for the purposes of the PLL. It uses blocks as functions and mathematical operators. Implementing the functional blocks of the PLL was a matter of translating the MATLAB code to blocks in Simulink. The model can be seen in Appendix A.

The Simulink model was built using the same parameters as the MATLAB simulation. As expected, its performance was identical to the performance of the MATLAB coded model, thus no results will be provided here.

3. Fixed-Point Model

The point of this thesis was to establish the plausibility of a fixed-point PLL in software and to determine what type of performance can be expected from such a system. This was to be achieved by building a model and running simulations to analyze the results. In building the fixed-point model, the same PLL parameters as the floating-point model were used.

As will be the case throughout the fixed-point process, the design problem faced at the input and output of each block is determining an appropriate scaling for the fixed-point number so that a large enough range is used, without introducing unacceptable inaccuracies due to a lower precision. As will be explained in Chapter 4.C, a 16-bit fixed-point implementation was used. Essentially, the reason for this was while an 11-bit representation was sufficient to run correctly, typical FPGA's and DSP chips use 8-bit, 16-bit, or 32-bit fixed-point arithmetic. The 8-bit representation didn't give enough precision for the system to lock with any reliability, thus 16-bits were used.

The procedure to convert the floating-point Simulink model to a fixed-point model involved looking at each block and determining the range of the signal at the input of the block and the range of the signal at the output of the block. With this information, a scaling appropriate for the range can be determined and, subsequently, the precision can be found at that scaling. If the precision was too large, an alternate method must be used, or an increase in the number of bits used.

To begin, the input signal to the PLL needed to be converted to its fixed-point equivalent. Recall the input signal was normalized to ±1. From Table 1, the best fit for this range of numbers is using a scaling of 2^{-15}. Because the range at this scaling is exactly the expected range of the input, any noise could be amplified, hence a slightly larger range was used, specifically a scaling of 2^{-14}. This gives a precision of $6.1035e^{-5}$, which is still below the noise of the input signal, thus is acceptable resolution.

With the input signal converted to fixed-point, the logical next step was to convert the PD to a fixed-point equivalent. The block diagram of the PD is shown in the Appendix in Figure A2 (duplicated on the next page for convenience). Tracing the connections, one can see that this is equivalent to the MATLAB code implementation in Appendix A. The PD takes the input signal

and multiplies it by the in-phase and quadrature-phase components of the NCO. In the block diagram of Figure A2, *In2* (short for input 2) is the phase, $\theta_2(n)$, which goes into a look-up table for sine and cosine producing the in-phase and quadrature-phase signals. The multiplication mentioned above multiplies two signals with a range of values between –1 and +1, thus the range of the output of this block must be within the range of –1 and +1. As can be seen in Figure A2, below the multiplication blocks Product1 through Product4, one can see the scaling is again 2^{-14}, reflecting the range required at the output of this block.

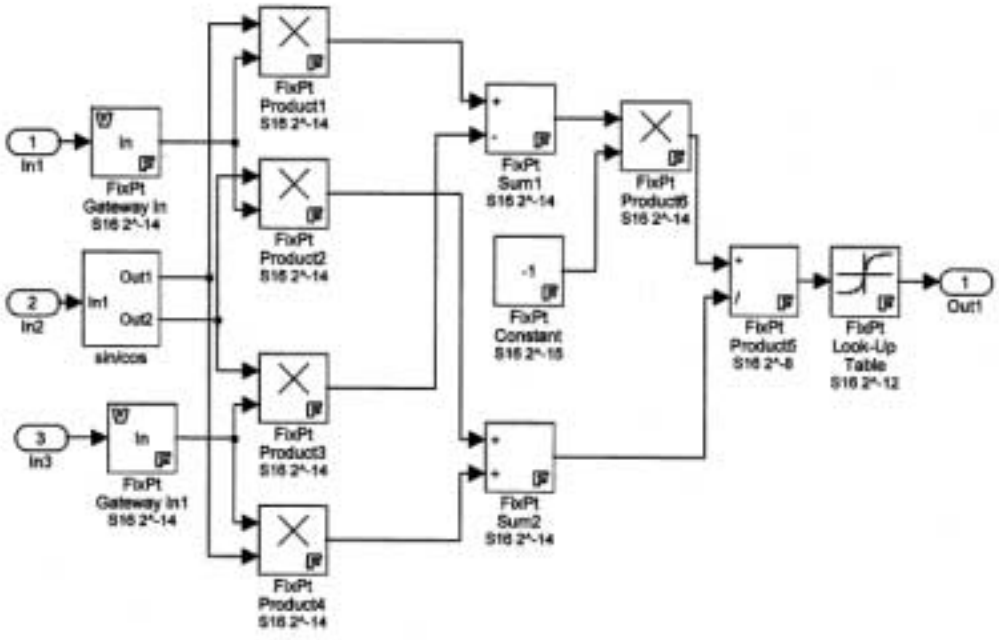

Figure A2. Phase Detector Simulink Model. (Duplicated From the Appendix for Ease of Reference.)

To belabor the point, the next blocks in the PD, Sum1 and Sum2, simply add or subtract the output of the multiplication blocks, producing a range of –2 to +2, which is the range of the scaling 2^{-14}. As can be surmised, this process of determining the range at every block needs to be done, and appropriate scaling of the output determined. Care must be taken in ensuring that when applying the best possible range to the output, a resolution problem isn't introduced. This issue will be discussed in Section 4.C. and is the reason an 8-bit implementation was not possible. The full block diagram of the Simulink PLL is included in Appendix A, with scaling at each block indicated.

The LF was designed using the $1/z$ block. Referring back to the z-transform of the difference equation established for the MATLAB simulation and rearranging equation (4.4),

$$U_f(z) = -a_1 z^{-1} U_f(z) + b_0 U_d(z) + b_1 z^{-1} U_d(z), \tag{4.9}$$

where z^{-1} is the unit delay operator. This equation is the one used in the Simulink model for the loop filter.

The NCO was implemented similarly using constant blocks and multiplier blocks, ensuring the range of the inputs and outputs were scaled properly. The complicated circuitry in the NCO is due to the fact that θ_2 grows each increment and must be reduced to $-\pi$ to π. This was done using a comparator block and a switch. The output of the switch is the top input, which is simply θ_2 for $\theta_2 < \pi$. If $\theta_2 > \pi$, the comparator resulted in zero, the switch outputs its third input vice its first, which is $\theta_2 - 2\pi$.

B. PERFORMANCE AND ANALYSIS OF SIMULINK PLL MODEL

The performance of the model was consistent with the MATLAB simulation considered in Subsection 4.A.1.c. For high SNR values, the fixed-point model performed identically to the floating-point model in lock range, lock time and pull-in time. The error of the output signal due to the quantization of the fixed-point signal became evident in the fixed-point model. This error in effect reduces the SNR of the output signal.

1. Performance Measures of the Fixed-point Model

Figure 13 shows the time response of the output of the PD, corresponding to the error in the PLL. It is clear that the error, θ_e, dies to zero, thus confirming the PLL achieves lock. To generate this plot a reference frequency of 2100 Hz was used.

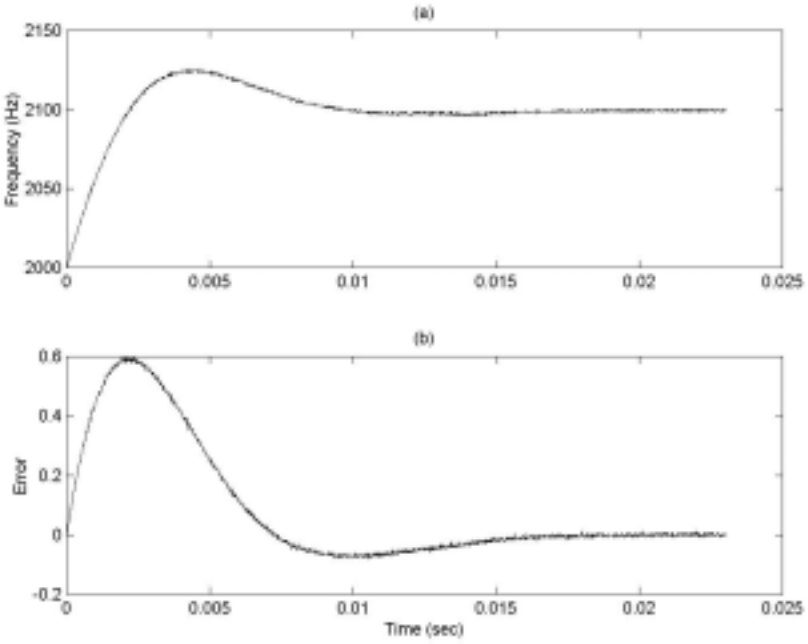

Figure 13. (a). Plot of Fixed-point Model NCO Output. (b). Plot of Fixed-point Model Error.

To establish the performance of the PLL, the theoretical values of the performance measures will be computed and then compared to the results obtained by the simulation. From Section 2.C, five key performance measures were derived. These are lock-range, lock-time, pull-in range, pull-in time, and pull-out range. For the parameters used in the simulation, the following theoretical values for the fixed-point PLL are calculated.

1. $\Delta\omega_L = 200\pi$ rad/s.
2. $T_L = 0.0120$ s.
3. $\Delta\omega_p = \infty$.
4. $T_P = \frac{\pi^2}{16}\frac{\Delta\omega_0^2}{\zeta\omega_n^3}$.
5. $\Delta\omega_{PO} = 1508$ rad/s.

For the testing of the fixed-point PLL, a worst-case scenario input was adopted. With this in mind, the random phase was dropped in favor of an artificially placed phase offset of 90 degrees. This represents the worst possible phase offset and, thus, the most conservative measures of performance.

a. Lock Range

The lock range of the PLL model is the range of frequencies at which the system locks in a single operating cycle. This corresponds to one period of the natural frequency ω_n. To determine this value experimentally, the input signal frequency was iteratively increased until the system did not lock in a single cycle. This established a bracket, or upper and lower bound on the lock range. By successively narrowing this bracket down, the lock range was determined. Using this procedure the lock range was found, $\Delta\omega_L = 518\pi$ rad/s, a value over twice as large as the theoretical value of 200π rad/s.

While a large lock range is desirable, some explanation as to why it is larger than predicted is required. There are two reasons this is the case. First, when determining the lock range, an approximation on the gain of the loop filter at the frequency offset was used. The assumption was that the lock range was larger than the corner frequencies of the two time constants, and thus the corner frequency was used as an approximation. The lock range is indeed larger than the corner frequencies, but with the non-linearity of the loop filter transfer function, a tighter bound is difficult to calculate, thus the overly conservative loop filter corner frequency is used.

Second, the fact that noise was not a factor in the design of this PLL, the noise bandwidth is very small, which means the noise is not hampering the locking of the device. This results in a larger lock range than theoretically calculated.

b. Lock Time

The lock time can be visually seen in the plot of Figure 13, but for a more accurate determination, the discrete output values were analyzed and the lock-time determined to be when the output entered and remained within 2% of the final value of the system. This is an accepted percentage for the settling time of a system in control theory. This is equivalent to the time required for the error to die down and remain below 0.02. However, the actual analysis is more involved. With the introduction of noise, a large noise value at any given sample can throw the output outside the 2% range, even though the output due to the system is within the 2% range. Examining the numerical values at each sample and finding where the output stayed below 0.02, the lock-time was found to be at 0.0128 s. Comparing this value to the theoretical value of 0.0120 s, the fixed-point representation does not degrade performance in this area significantly. It must

be understood also that the theoretical value of 0.0120 is assuming no noise, and that any significant spike in the noise at the output of the system could effect the settling time of the system when analyzing the output at each time sample. However, it should be noted that running the simulation several times, the above settling time of 0.0128 was the average of all simulations.

c. Pull-in Range

The pull-in time for the second-order PLL is theoretically infinite, but an analysis of the pull-in range of the fixed-point system needs to be done. At some point, the frequency offset will get so large that the fixed-point precision or range will be exceeded and critical errors will be introduced. Incrementally increasing the input frequency, the fixed-point PLL eventually obtained lock for a max frequency of 2830 Hz. This value represents a 41.5% error from the center frequency of the PLL.

The limitation seems to come from the loop filter. The range or precision, depending on the scaling used, is exceeded at the output of the loop filter, causing the input to the NCO to be inaccurate. The same limitation was discovered when attempting to develop an 8-bit fixed-point PLL and will be discussed in more detail in Section 4.D.

d. Pull-in Time

The pull-in time, T_P, of the system is dependant on the frequency offset from the center frequency. Realize that if the frequency offset is inside the lock range, the lock time is used and is the same for any frequency offset, as long as it is smaller than the lock range. The pull-in time varies with the magnitude of the offset. To get an idea of the performance as related to pull-in time, five input frequencies were used and compared to the theoretical values for those frequency offsets. The settling time again was calculated using a 2% tolerance. Table 2 shows the results for the five frequencies. The pull-in time is given in seconds for the theoretical as well as the experimental and the frequency offset is given in Hz. An initial offset of 300 Hz was chosen due to the fact that from the analysis of lock-range, $\Delta\omega_L = 259$ Hz and, thus, a larger value than this was needed. The pull-in range of the system was determined to be 830 Hz, thus only values up to 800 Hz were included.

Δω [Hz]	Tp [s] (theoretical)	Tp [s] (experimental)

300	0.0254	0.0210
400	0.0452	0.0305
500	0.0707	0.0446
600	0.1018	0.0640
700	0.1385	0.0836
800	0.1810	0.1114

Table 2. Pull-in Times for Various Frequency Offsets

The disagreement between the theoretical and experimental values, especially as the frequency offset increases, cannot entirely be explained by the approximations used in deriving T_P. Another explanation lies in the discrete quantization of the fixed-point PLL. The output of the PD, or the error of the system, has a certain precision. The range of numbers required at that block, as described previously, determines this precision. If the value at the output of the PD is smaller than this precision, the value is rounded to zero. By the nature of the second-order PLL, any time two successive outputs fall within the precision limits, the output remains at zero. The above theoretical values assume an analog PLL, where precision is infinite. A fixed-point software PLL has quantization error, which in this case, output a zero even though the real-world value is larger than that.

e. Pull-out Range

The pull-out range is an important figure because it allows the user to determine how large a frequency step can be supplied to the input without unlocking the system. This assumes the system is already in lock and a frequency step occurs, as in an FM system. This value was tested by giving the PLL a signal, allowing lock to occur, then stepping up the frequency until it unlocked. Simulations of the fixed-point PLL agreed with the expected figure, which was obtained through computer modelling in Reference [5]. The fixed-point PLL experimentally had a pull-out range of 1501 rad/s or 239 Hz, compared to the theoretical value of 1508 rad/s, or 240 Hz. Recall that the lock-range of the system was 259 Hz, thus if a frequency step larger than the pull-out range, but less than the lock range, the system will again lock in the lock-time, or 0.0120 seconds for this system. If the frequency step is larger than the lock-range, the system will acquire lock using the longer pull-in time as described above.

f. Frequency drift

The magnitude of a frequency drift is of interest because, for a software radio implementation and downloadable software radio packages, satellite communication is required. As neither satellites nor users are stationary, a frequency drift can occur due to the Doppler shift in frequency. The Doppler shift is governed by two parameters: the velocity and direction of travel relative to the antenna, and the wavelength of the signal. Specifically, the Doppler shift in frequency is,

$$f_d = \frac{v}{\lambda}\cos\theta . \qquad (4.10)$$

where $v =$ velocity, $\lambda =$ wavelength or c / f_C, c=speed of light , and $\theta =$ direction of travel from the antenna. For a frequency drift to occur, the satellite would need to either be changing velocity or changing direction. With an orbital satellite, the direction of movement is constantly changing. Therefore, an idea of what kind of drift the PLL can handle is an interesting figure of merit.

From Subsection 2.B.1.c, the rate of change in a frequency ramp that will cause a system to unlock is governed by Equation (2.24). To test the fixed-point PLL model, a frequency ramp was applied to the input after the system was allowed to lock. The slope of this ramp was increased in each run until the system was observed to unlock and to be unable to track the frequency change. The rate at which the PLL was unable to track the frequency ramp was much smaller than that predicted by Equation (2.24), but this value was assuming no noise and an analog system. Because of the non-linearity of a frequency ramp, noise and quantization error are magnified. The results of the fixed-point PLL are $\Delta\dot{\omega}_{MAX} = 15,000$, compared to the theoretical value of 130,000. The PLL could maintain lock up to this rate of change for about 0.25 seconds, corresponding to about 30 cycles of the system. From 0.25 seconds on, the large frequencies were out of range of the fixed-point representation. The PLL maintains lock longer for a more gradual frequency ramp slope (see example below). The limiting factor is simply the range of frequency that must be covered for the case of a frequency drift. For a fixed-point PLL with a center frequency of 2000 Hz, the system unlocked when the reference frequency reached around 3300 Hz. For smaller sloped frequency ramps, the system remained locked longer, but once it reached the upper bound of 3300 Hz, the system fell apart.

To understand if this is a limitation for a real world application, an example of a moving user is given. There are two ways a frequency ramp can be seen at the receiver of a PLL

due to the movement of a user. The first is if the user is accelerating in speed. The second is if the user is changing direction. The second case will be considered first. The largest change in frequency due to a change in direction occurs when θ goes from 1 to –1. This represents a user turning around and moving in the opposite direction. The user will be assumed to travel at a constant velocity of 60 mph (26.82 m/s). If the user is assumed to turn-around on a dime, the result is a frequency step. The performance of a frequency step was covered in Subsection 4.B.1.e, therefore a circular path of the user will be assumed, which represents a constant change in angle, thus a frequency ramp as the input signal of the PLL. As shown in Figure 14, the user makes a tight circle with a radius of 10 m. To get anything but negligible values, a high frequency signal of 100 MHz is used. At 26.82 m/s the turnaround would take them 2.34 seconds. The value θ in Equation (4.10) goes from 0 to π in 2.34 seconds. This represents a slope of the frequency ramp of 48 radians. A slope of 48 radians would take 170 seconds, or almost three minutes for the PLL to unlock. The frequency ramp is only applied for 2.34 seconds. The system can easily handle a frequency ramp of this slope, even with an elevated frequency of 100 MHz.

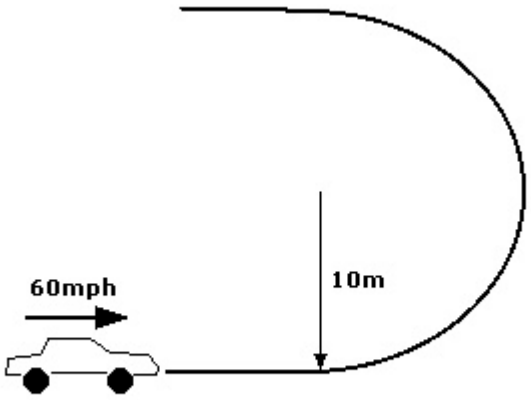

Figure 14. Vehicle Driving 60 mph Around a Tight Curve to Illustrate how the Doppler Shift can Cause a Frequency Ramp of a Signal.

The second case is when the user is accelerating. Suppose the user goes from a dead stop to 60 mph in 3.1 seconds (the fastest production car ever built is the Porsche Carrera twin turbo which was tested at 0-60 mph in 3.1 seconds). The resulting frequency ramp has a slope of 18.12 radians. For a slope of $\Delta\dot{\omega} = 18.12$ rad/s^2, the PLL would take 450.78 seconds to

unlock. Even in an accelerating jet, the frequency ramp caused by acceleration is well within the limits of the PLL

For normal operating environments, the PLL can be expected to stay locked for a real-world frequency ramp. If an environment where the frequency ramp is too large, or applied for too long is encountered, a second PLL with an appropriate center frequency could be used.

2. Analysis of PLL

The performance measures of the PLL discussed in the previous section give a good idea of the capabilities of the fixed-point PLL, but two further items concerning the PLL need to be analyzed: the error at the output due to the fixed-point representation, and why an 8-bit system failed. The first will be examined first as it will shed some light on the second.

a. Errors in the Output of the PLL

To get an idea of the errors expected at the output of the PLL, an analysis of the errors that could possibly accumulate at each stage of the PLL is required. A full discussion of error analysis and propagation of errors is beyond the scope of this research; however, to determine the size of errors at the output of the PLL, some error analysis is required. This will be done by looking at the precision at each stage of the PLL and from that precision a worst-case error can be determined. By finding the largest possible error in magnitude at each block, the error of the sinusoidal output of the NCO is determined. Errors at the output of each block are due to two factors. The first is the error at the input of the block. The second is the error due to the fixed-point representation and arithmetic operation of the block. Adding these two error results in the error at the output of the block.

The input signal was assumed to be normalized to amplitude of ± 1, thus the best scaling of the fixed-point number that includes that range is $S = 2^{-14}$ (see Chapter 3 on scaling of fixed-point numbers). This results in a precision of $6.10E-5$. Recall that precision is defined as the distance between two successive representable numbers. If a "round to nearest" implementation is used, the largest possible error that can occur is half of this precision or $3.05E-5$. The convention of $e_r = \text{accumulated error}$ will be adopted for the remainder of this

section. With this notation, when the real world signal is converted to a fixed-point value, $e_r = 3.05E-5$.

For the PD component of the PLL, the input signal is multiplied by the output of the NCO. The initial value of the NCO is zero, but it is sinusoidal in nature. The assumption will be made here that the initial error in the output of the NCO is due only to the precision of the fixed-point number representation. The NCO sinusoid output also has amplitude ± 1, thus uses the same range and precision. For multiplication, the error in the product is the sum of the error in the two multiplicands. The error of both multiplicands is the same, thus after the multiplication of the reference signal and the NCO, the error due to the input becomes $2*3.05E-5 = 6.1035E-5$. Adding the error due to the fixed-point operation yields a value of $e_r = 9.1552E-5$.

The next block adds the in-phase and quadrature-phase components in the PD. Again, finding the error due to the inputs and adding this value to the error due to the fixed-point operation determines the error. For addition, the error due to the inputs is the sum of the errors. This results in a value of $1.8310E-4$. Adding the resolution error, the output of the summation has a possible error of magnitude $e_r = 1.8310E-4+3.0517E-5 = 2.1362E-4$.

The next step of the PD was to divide the in-phase and quadrature-phase signals. To determine the error at the output of this operation, assume first that the division operation is perfect and that the output error is caused only by the error at the input. This being the case, the error propagates the same as multiplication. The resulting error is the sum of the errors at the input. Therefore the error at the output due to error at the input is $2*2.1362E-4 = 4.2724E-4$. Unfortunately the errors of this operation do not stop here. If the denominator in this operation is small compared to the numerator, a large number will result. In fixed-point arithmetic, this means a larger range of numbers needs to be represented. For the division in the PD, the range of numbers was determined to be between ± 128. To get this type of range, a scaling of $S = 2^{-8}$ is needed, which results in a precision of $3.906E-3$. The worst-case error at this stage is the error due to the input plus the error due to fixed-point scaling. Adding these two errors together, $e_r = 2.3804E-3$.

The next block of the PD is the $\tan^{-1}(x)$ function. This function is asymptotically bounded by $\pm\pi/2$, thus a scaling of $S = 2^{-14}$ is used. Because this is a non-linear function, errors do not work exactly the same. The error will be maximized where the function has a maximum derivative. This is at $x = 0$. To find the error at the output, the function must be applied to the error at $x = 0$. More specifically, if the error at the input of this stage is $e_r = 2.3804E-3$, the represented value of the fixed-point system could be 0 or $2.3804E-3$. If the $\tan^{-1}$ function in the system was perfectly continuous, the error at the output could be as large as

$$\begin{aligned} e_r &= \tan^{-1}(0) - \tan^{-1}(2.3804E-3) \\ &= 2.3804E-3. \end{aligned}$$

Including the error due to the fixed-point representation of $\tan^{-1}$ function, the error is $e_r = 2.3804E-3+3.0517E-5 = 2.4109E-3$. As this is the last block of the PD, the total error of the phase detector component is $e_{r,PD} = 2.4109E-3$.

The loop filter component is the biggest contributor to error in the system. This is because the loop filter lead-lag constants are large, thus a larger range is needed, reducing precision. The input to the loop filter is the output of the $\tan^{-1}$ function, which has a range of $\pm\pi/2$. This signal is multiplied by the constants b_0 and b_1. For the PLL model, these two values were 635.14 ± 0.015625 and -621.43 ± 0.015625, respectively. The output of this multiplication needed to include the entire range of $\pm(\pi/2)(635.14) \approx \pm 997$. This required a scaling $S = 2^{-5}$, which has a precision of 0.03125 or output error of 0.015625. Adding the error due to the inputs of the multiplication to the error from fixed-point multiplication, the accumulated error becomes $e_r = 3.3661E-2$ on each multiplication.

The error gets worse when the addition in Equation (4.5) of the loop filter is applied. This addition has three terms in it. The first term in Equation (4.5) has the same range as the output of this addition. Because of the feedback of the system, a range of ± 5000 is needed. This results in a precision of 0.25 or error of 0.125. The error in the three terms are added together then added to the precision of the output. The total error of the loop filter then becomes,

$$e_{r,LF} = 2*3.3661E-2+0.125+0.125 = 0.2837. \qquad (4.11)$$

The first block in the NCO performs the operation

$$\theta_e(n) = \omega_0 T_S + u_f(n) T_S . \qquad (4.12)$$

The first term in this sum is a constant. Evaluated to be 0.62831, it is the center frequency of the PLL multiplied by the inverse of the sampling frequency. For this PLL model $T_S = 1/F_S = 5E-5$. The second term varies with the signal $u_f(n)$; hence a range of numbers is required vice the constant first term. The range of $u_f(n)$ is the range of the output of the loop filter, which was previously determined to be ± 5000. Thus the range of the second term in the sum is $\pm 5000 * T_S = \pm 0.25$. For this small range, a scaling of $S = 2^{-15}$ is used, which increases precision. To determine the error at the output of this multiplication, the error due to the precision of the output is added to the largest possible error at the input, $e_{r,LF}$ multiplied by T_S. Thus the error at the output becomes,

$$e_r = e_{r,LF} * T_S + \frac{1}{2} 2^{-15} = 2.9442E-5 .$$

Adding the first term in the sum of Equation (4.12), a new range is needed due to the larger value of 0.62831. This larger range results in a scaling of $S = 2^{-14}$. The error due to this precision is $3.0517E-5$. Adding the error in the fixed-point representation of 0.62831, the error in the second term of $2.9442E-5$ and the error due to the fixed-point operation yields an error of $e_r = 9.0476E-5$.

The final possible introduction of errors in the system is due to feedback property of the NCO and the continual scaling of the output $\theta_2(n)$ to keep it between $-\pi$ and π. The output phase of the NCO equals, $\theta_2(n) = \theta_2(n-1) + \omega_0 T_S + u_f(n-1) T_S$. If $\theta_2(n)$ is assumed to remain between $-\pi$ and π, $\theta_2(n-1)$ must be in this range as well. The appropriate scaling for this sum is $S = 2^{-13}$. This results in an error of $6.1035E-5$. The last two terms of the sum have a total error of $9.0476E-5$. Adding the errors due to fixed-point scaling and errors due to the feedback input, the resulting error of $\theta_2(n)$ is equal to $e_r = 9.04756E-5 + 2 * 6.1035E-5 = 2.1254E-4$.

To keep $\theta_2(n)$ bounded, the algorithm adopted was to subtract 2π whenever it got larger than π. To get an accurate error of the system, this subtraction must be taken into account, despite the fact that it occurs infrequently. To subtract 2π from the phase whenever it grows larger than π requires converting 2π to a fixed-point value. The error in this conversion is $6.1035E-5$. Again adding the error due to the inputs to the error due to the fixed-point operation, the total error of the NCO is determined to be,

$$e_{r,NCO} = 3.3462E-4. \qquad (4.13).$$

The error determined thus far is the error in the phase of the NCO. The output of the NCO is a sinusoid (actually two sinusoids, the in-phase and quadrature-phase components of the signal, but the error will be the same for both) with amplitude one. The scaling used for this range is $S = 2^{-14}$. To determine the output error of this sinusoid, the error at the input must be applied to the sinusoid function where its derivative is maximum. For the sine function, this is at zero, thus the error at the output due to the input error is $\sin(3.3462E-4) - \sin(0) = 3.3462E-4$. This value must be added to the error of the sinusoid due to its precision. With a scaling of 2^{-14} a fixed-point quantization error of $3.0517E-5$ is found. Adding these two errors together, the total error of the system is determined to be

$$e_{r,TOT,1} = 3.3462E-4+3.0517E-5 = 3.6513E-4. \qquad (4.14)$$

The subscript (1) refers to the error after the first iteration of the feedback loop. At the beginning of this analysis, the NCO error was multiplied by the reference signal input. Because this is a feedback loop and a start point had to be determined, the NCO input error was assumed to be zero. Now that a figure for the error in the NCO has been found, this assumption requires correction. If the error in the NCO signal is now added and the whole error analysis repeated from the PD, the error output after the second iteration is,

$$e_{r,TOT,2} = 4.5743E-4 \qquad (4.15).$$

This error analysis shows that as the feedback system continues to operate, the error can grow. At this point an idea of how fast errors can propagate and grow in the feedback system is determined. By defining,

$$\Delta e_r = e_{r,i} - e_{r,i-1}, \qquad (4.16)$$

the Δe_r for $i = 1$ is $\Delta e_r = 9.2296E-5$. This does not mean that the error *will* grow at this rate, because errors can cancel themselves out. However this is the maximum rate at which they could be expected to grow.

Figure 15 shows the output of the fixed-point NCO subtracted from the output of the corresponding floating-point NCO. This difference represents the error in the fixed-point implementation. The error can be seen to oscillate, with a maximum value of around $2.4E-3$.

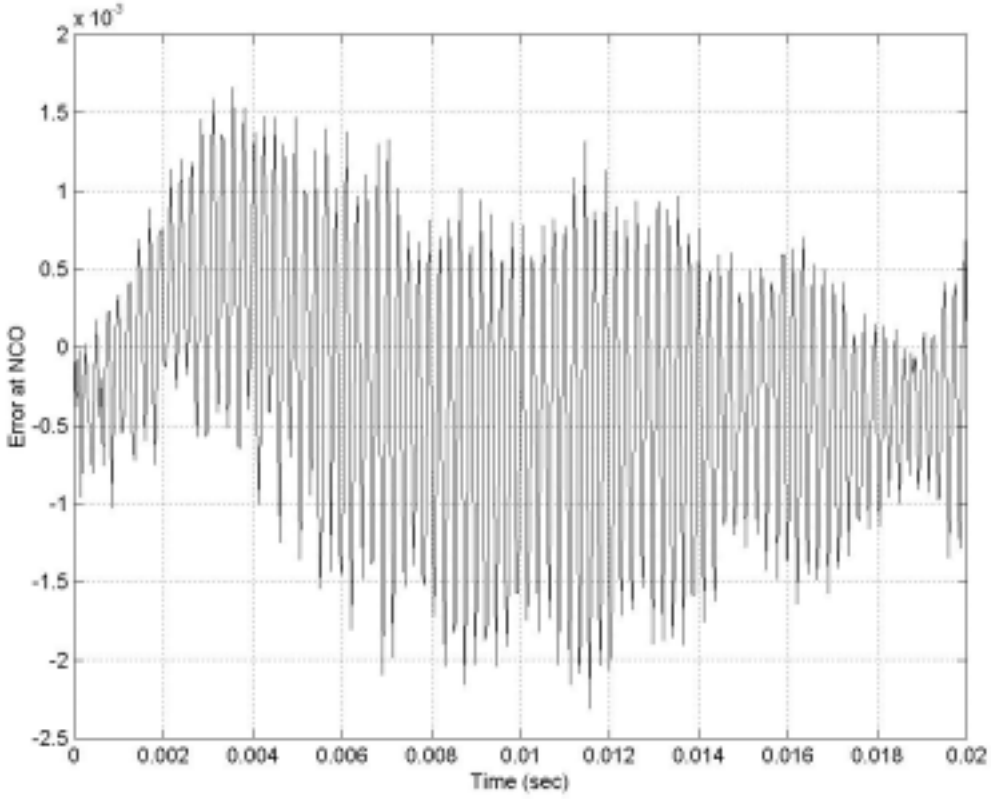

Figure 15. Error Plot of the Output of the Fixed-point NCO versus Time

How does this affect the performance of the PLL? For the $\mathrm{SNR}_i = 15\mathrm{dB}$ used for this simulation, it essentially does not. Using a maximum error of $2.4E-3$, and the input SNR of 15 dB, the error reduces this by 0.164 dB. The new . This is not a problem for the PLL considered. The PLL locks consistently and as shown in the sections above, within expected theoretical limits. The quantization error of a fixed-point system could be a concern if

implemented in a low SNR system, but for communication purposes, with SNRs over 10 dB, the reduction will not impact the performance of the PLL.

b. The 8-bit PLL

An 8-bit PLL was attempted, but as can be surmised, the 8 bits did not give a large enough range or a high enough resolution. When one parameter was fit to the dynamics of the PLL, the other was too small, causing critical errors. This last section examines why this was the case.

The only component of the PLL that could not be made to fit an 8-bit representation is the loop filter. This is because in this component the largest ranges are encountered. Recall from Equation (4.5), the output of the loop filter is described by

$$u_f(n) = u_f(n-1) + b_0 u_d(n) + b_1 u_d(n-1),$$

where $b_0 = 635.1475$ and $b_1 = -621.4397$.

The initial cycle of the PLL sets $u_f(0) = u_d(0) = 0$. This means $u_f(1) = 635.1475 u_d(1)$. Assuming the worst possible phase offset between the reference and the signal sent from the NCO of $\pi/2$, $u_d(1) = \pi/2 \approx 1.5$. From Equation (4.5), $u_d(1) = 635.1475 * 1.5 \approx 982$. If a phase offset of $-\pi/2$ is used, $u_d(1) \approx -982$, thus at a minimum this range must be included. To achieve this large range with 8-bits requires a scaling of $S = 2^3$. The corresponding precision for this scaling is 8. The second iteration comes along and, because a correction has been made to the NCO output, the phase error is slightly less, around 1.4. Plugging in the formula for u_f,

$$u_f(2) = 982 + 933 - 932 = 939.$$

It is important to notice that the last two terms of this sum add to 1, but with a precision of 8, the result is zero, thus $u_f(2) = 982$. No correction from the second iteration is made. The PLL cannot lock as the corrections to the phase of the NCO frequency cannot be realized due to the large range, coupled with the need for high precision.

Numerous methods were tested to correct this error, but with little success. The sum was broken down into numerous sums, so that a higher precision could be obtained, but the initial loop made this approach unrealizable. A second more promising approach was to use two 8-bit numbers in the loop filter, similar to a double floating-point number. Unfortunately, Simulink does not make it easy to implement this type of representation and the resulting circuitry and eventual hardware implementation complicated the system more than using 16-bit circuitry.

c. Effects of Changing PLL Parameters

If not already realized, it should be mentioned that the scaling of the fixed-point numbers is a crucial step in developing a fixed-point model. The range of the numbers and precision must be considered and evaluated at each step, and when the two conflict, priorities must be set to determine a solution.

It must be realized that changing parameters can result in changing the range or precision of certain points in the model. This is especially true when changing the sampling frequency of the system. The sampling frequency can have a large range in itself and thus a dramatic effect on the PLL operation. For the model considered here, a sampling frequency of 20 kHz was used. Reducing the sampling frequency to 14 kHz, the model reluctantly locks, but jumps are seen at the output, indicating unlock at certain places due to out of range errors. Drop the sampling frequency down to 12 kHz and the model is nearly useless as a PLL. The PLL does seem to lock, but at the steady state, is unlocked as often as locked. The output signal of the system is extremely jumpy, and useless in a communications application.

Changing the lock range of the system can also have dramatic effects on the system. Once the lock range is given, the natural frequency is calculated from the given lock range. This value is used to calculate nearly all of the parameters in the PLL. Changing the lock frequency can cause an avalanche of changing values, resulting in a different scaling requirement for the fixed-point representation of values.

E. CONVERSION TO HARDWARE

To complete the analysis of the fixed-point PLL simulation, converting this model to a hardware implementation, either on a DSP or FPGA chip needs to be mentioned. Mathworks has a product that works in conjunction with Simulink to produce C/C++ code from Simulink models.

It provides a debugging feature and code optimization package. This software package is called Real-Time Workshop. At the time of this writing, the software package was not available to the author, thus testing the conversion to code was not possible. The literature suggests that this package will translate a Simulink model, whether discrete, continuous, or fixed-point, into a portable, working program.

Once this program, in C/C++ or Ada, has been developed, converting it to a VHDL language is simply a matter of syntax (HDL stands for Hardware Description Language). This type of language was adapted as the need for a language that could be used by a program-controlled machine for generation of final hardware was recognized. Several commercial HDL packages are available today.

In this chapter, the procedure for designing the fixed-point PLL model was described. This was done by building a floating-point model and converting it to a fixed-point equivalent. Analysis was done by comparing the fixed-point simulation results to those obtained from the floating-point. Finally an error analysis was done to determine how large errors could be expected to grow due to the fixed-point arithmetic quantization error. The final chapter deals with capabilities, limitations, and future research for a FPGA based PLL.

V. CONCLUSIONS

The objectives of this research were to build a fixed-point PLL and analyze the performance of the model. The was accomplished by building a floating-point PLL model and converting it to fixed-point arithmetic and comparing the results of the two simulations. This Chapter discusses the capabilities and limitations of the fixed-point model, and concludes with recommendations for further research in this area.

A. CAPABILITIES AND LIMITATIONS

The fixed-point model developed was able to lock onto a given reference input signal with comparable performance to the floating-point or analog equivalent. Lock-ranges and lock-times in the model were equivalent. The model proved to be able to track a drifting frequency, lock onto a frequency step larger than the theoretical value, and a phase step with accuracy and within theoretical limits. As a fixed-point implementation, it is a viable option for use on a FPGA or DSP chip, fulfilling the requirement for a PLL in a software radio package. The reason this is the case is because the fixed-point operations are much faster than the IEEE floating point standard, allowing real-time radio in a downloadable software-like format. The second major advantage of the fixed-point implementation is the memory storage required. Because the range of the numbers is known, 16-bits can be used and scaled appropriately to represent each number as opposed to 32-bits or 64-bits of floating point. This reduces the amount of memory required by 50% or 66%, a valuable commodity on a small DSP chip.

The cost of fast operation and memory savings is a trade-off in precision and range. Using only 16-bits means the range of numbers or the precision much be sacrificed. The limitation of the fixed-point model is the parameter specific operation of a given model. The point behind using 16-bits and a fixed-point operation is that the range of numbers is known, and the window of numbers that 16-bits can represent can slide to accommodate that range. The other side of this is that, if parameters are changed and that window requiring representation changes, the fixed-point model could fail. To have a completely portable model, the scaling of the fixed-point numbers needs to be variable. By allowing the scaling to be variable, any change in parameters

can be accounted for by changing the scaling, hence sliding the window of fixed-point representation to match the changing range.

The 16-bit fixed-point model introduced error into the PLL that otherwise would not be there. This is seen in the quantization error in representing a number. To speed up arithmetic operation, fixed-point numbers are represented in a way that makes binary arithmetic – the arithmetic a processor uses – much faster. This representation is not optimal for representing real-world values. As was shown in Chapter 4, this error resulted in a reduction of the SNR of around 0.16 dB. For radio communication purposes, this is not an obstacle, but for applications using a small SNR, limitation on fixed-point representation needs to be considered.

As mentioned numerous times, fixed-point implementation has a smaller range of representable numbers, given a certain precision. For the loop filter used, a certain precision is required to make sure small additions have an effect on the output. This results in the precision being the priority and, hence, the range suffers. Two limitations are the cause of this limited range.

First, an 8-bit PLL was not realizable. By using 8-bits, the range was decreased to the point that numbers were over 50% outside the representable range. This produced critical failure errors in the feedback loop, preventing the PLL from locking. Several solutions were attempted and could hold promise, but a 16-bit fixed-point implementation was preferred.

The second limitation of the reduced range is that the pull-in range of the system was reduced. While a second-order analog or floating-point PLL with an active P-I loop filter should have unlimited pull-in range, the fixed-point counterpart did not. This was due to the range restrictions imposed by the fixed-point representation. Because only a limited range could be represented, when a frequency outside that range was encountered, the value could not be approximated in the fixed-point scaling. This inevitably prevented the system from locking.

B. RECOMMENDATIONS FOR FURTHER RESEARCH

The main area for further research is to actually build a phase lock loop on an FPGA or DSP chip. This step will have two steps. First, the PLL model must be converted to useable

code. The Mathworks Real Time Workshop is ideal for this purpose. Research into this program and the how code is produced from a Simulink model needs to be done. This research would require understanding how different parameters in Real Time Workshop affect the generated code, and how to optimize the final program.

The second step would involve translation to an HDL . While there are several commercially available HDL's, including Interactive Design Language (IDL) from IBM, Instruction Set Processor Specification (ISPS), Test Generation and Simulation (TEGAS), Texas Instruments HDL (TI-HDL), and ZEUS, created at General Electric Corporation, the most promising language was developed in 1983. Contracted by the DoD, VHDL 2.0 was released in late 1983. In 1987, VHDL became the IEEE standard for HDL. It is sufficiently rich for designing digital systems and supports a hierarchical description of hardware. This means it can be used to describe programs from systems to gates, or even the switch level [7]. To continue this research, an understanding of an HDL and a conversion from the present Simulink model needs to occur. This process is not as easy as a simple conversion from C/C++. From the C/C++ code, a Preliminary Design Description (PDD) is typically generated prior to VHDL coding. Once an acceptable PDD is complete, VHDL coding can commence [3].

Analysis on what type of FPGA or DSP is needed to satisfy the requirements of a software radio needs to be done. This would include figures of merit such as speed and memory. If the radio requires GHz operation, but an FPGA can only handle MHz, a real-time system is not feasible. This is not the case, as "IF Sampling" can bring the operational frequency down, but research on the trade-off between high speed and affordability is required. Research into how much memory would be required to not only put the PLL on the chip, but the other component parts of a radio such as a demodulator, decoder, or delay lock loop (DLL).

C. FINAL COMMENTS

The final goal of this and related research is to build a software radio on a DSP or preferable an FPGA chip. The idealized software radio would allow the user the flexibility to download software packages related to their mission at that time. This platform would be a general-purpose radio. Instead of using several radios for different situations, modulation formats, or radio protocol, a single unit, capable of communicating with all formats by a

downloadable software package would be used. By doing all the demodulation, and signal processing in software, this goal can be achieved. A typical radio signal would be digitized at the antenna, or more likely, an analog frequency translator would be used to shift the RF signal to an intermediate frequency to reduce the sample rate requirement of the A/D converter. With the signal digitized, all radio processing function are done in software. An integral part of this radio function is the PLL. To realize these functions on an FPGA chip, fixed-point arithmetic is required. For reasons already specified such as memory space and speed, DSP and FPGA chips use fixed-point representation rather than floating point. Thus a fixed-point PLL model was developed.

The fixed-point PLL model simulated in this research performed comparable to an analog or floating-point equivalent. The fact that it was implemented using fixed-point arithmetic makes it ideal for conversion to an FPGA or DSP chip.

The limitations introduced by the fixed-point arithmetic must be considered in designing the final product. Understanding the nature of fixed-point scaling and how parameters can affect the ranges of an application are essential. For the model in this research, a 16-bit fixed-point representation was assumed. An 8-bit model would have been ideal, but the limitation on range and precision of only 8-bits caused a deferment to 16-bits. Fortunately, DSP and FPGA chips come in 16-bit and even as high as 32-bit on the commercial market, making the 16-bit implementation a practical solution.

APPENDIX A

This Appendix gives the code for the MATLAB floating-point PLL and the fixed-point Simulink model used for simulation.

Floating Point MATLAB code

For ease of reference, the variables used in the code will be matched to the notation used in the text.

- sigI = in-phase component of u_1.
- sigQ = quadrature-phase component of u_1.
- zeta = ζ.
- Wl = ω_L, Wn = ω_n.
- tau1 = τ_1, tau2 = τ_2.
- phi = θ_2.
- theta = θ_e.
- F1 = output frequency of u_2.

The following is the code generated for PLL model.

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%  ADPLL2 (All-Digital Phase Lock Loop)                                       %
%                                                                             %                    %
%  Implements a phase lock loop assuming a complex in-phase and               %
%     quadrature-phase input signal.  Adapted from Reference [1].             %
%                                                                             %                    %
%                                  Michael Johannes                           %                    %
%                                  12 September, 2002                         %                    %
%                                                                             %                    %
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

% Initialize signal parameters

Fc = 2100;              %  Input Reference signal frequency
U10 = 1;                %  Input Reference signal amplitude
U20 = 1;                %  Output NCO signal amplitude
F0 = 2000;              %  Center Frequency of NCO

Fs = 40000              %  Sampling freqency
Ts = 1/Fs;              %  Sampling duration

%  Construction of Reference signal sigI and sigQ

SNRdb = 15;             %  Input Signal to Noise ratio in dB
BW = 1000;              %  Prefilter bandwidth
pod = rand*360;         %  Randomize phase offset in degrees
por = pod*pi/180;       %  Convert random phase offset to rad/sec
t = 0:Ts:50/Fc;         %  Vector of time samples

%  Calculate noise power from SNR and prefilter bandwidth

sigma = sqrt(.5/(10^(SNRdb/10))/BW)
sigma = sqrt(.5/10^(SNRdb/10)/BW);

% Signal construction with noise added

sigI = U10*cos(2*pi*Fc*t+pi);
sigQ = U10*sin(2*pi*Fc*t+pi);
sigI = sigI+sigma*randn(size(t));
sigQ = sigQ+sigma*randn(size(t));

%  Calculation of PLL parameters

zeta = .6;              %  Damping factor
Wl = 200*pi;            %  Lock range defined according to expected
                        %     input characteristics

Wn = Wl/2/zeta          %  Natural frequency calculated from lock range
Kd = 1;                 %  Phase detector gain
Ko = 1;                 %  Loop filter gain

%  Loop filter constants and lead-lag gain constants calculated
```

```
tau1 = Ko*Kd/(Wn^2);
tau2 = 2*zeta/Wn;
a1 = -1;
b0 = (Ts/(2*tau1))*(1+1/(tan(Ts/(2*tau2))));       % Lead constant gain
b1 = (Ts/(2*tau1))*(1-1/(tan(Ts/(2*tau2))));       % Lag constant gain

%  Initialization for first cycle of feedback loop

phi = 0;                %  Output phase of the NCO
theta = 0;              %  Phase error of the two signals
Uf=0;                   %  Output signal of the loop filter
thetaI = 0;             %  Error of in-phase signal
thetaQ = 0;             %  Error in quadrature-phase signal
F1 = 2*pi*F0            %  Output frequency of NCO

% General case phase lock loop

for n = 2:length(sigI)

   phi(n) = phi(n-1)+F1(n-1)*Ts;    %  Phase error from current cycle
                                    %     added to the phase of the
                                    %     output signal

   % Subtract 2*pi whenever the phase is larger than pi to keep the
   %     phase bounded

   if phi(n) > pi
       phi(n) = phi(n) - 2*pi;
   end

    % Implementation of phase detector
    thetaI(n) = sigI(n)*cos(phi(n))+sigQ(n)*sin(phi(n));
    thetaQ(n) = sigI(n)*sin(phi(n))-sigQ(n)*cos(phi(n));
    theta(n) = atan(-thetaQ(n)/thetaI(n));

    %  Loop filter equation.
    %  Implements the transfer function     1+s*tau1
    %                                      --------
    %                                       s*tau2

    Uf(n) = -a1*Uf(n-1)+b0*theta(n)+b1*theta(n-1);

    %  Implements the equation for the NCO  Wo+Uf(n)
    F1(n) = 2*pi*F0+Uf(n);

 end   %  for loop
```

Fixed-point Simulink Model

The Simulink model was broken down into its component parts: Phase Detector, Loop Filter, and Numerically Controlled Oscillator. Figure A1 shows all components in the PLL system. Figures A2-A4 show the Simulink block structure for each component. The scaling of the fixed-point representation for each block is shown next to that block at the lowest level.

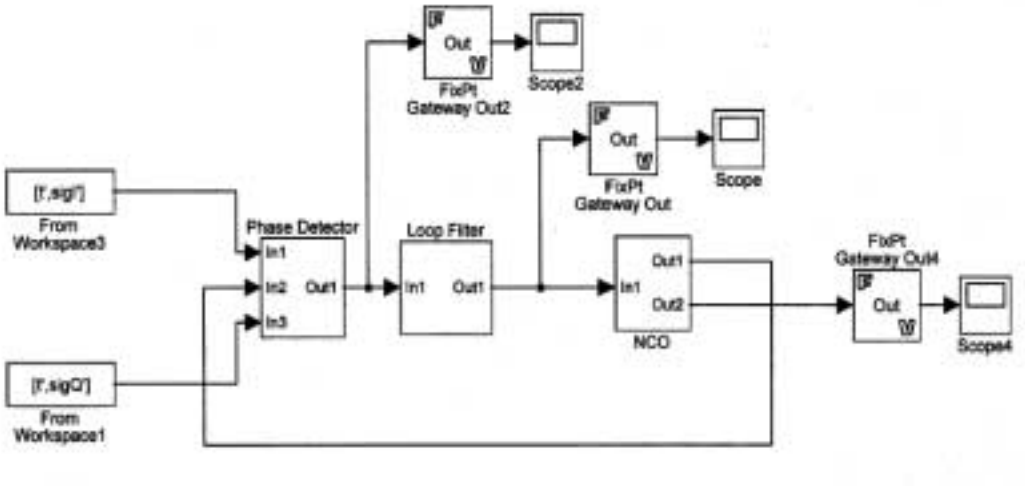

Figure A1. Phase-Lock Loop Simulink Model

The PD takes three inputs. The inputs *In1* and *In2* are the in-phase and quadrature-phase reference signals, depicted by the $[t', sigI']$ and $[t', sigQ']$, respectively. These signals are generated in the MATLAB workspace, and imported into Simulink using the *From Workspace* blocks. The t input of that block simply tells the model it is a sampled signal and the duration between samples. The input *In2* is the phase error feedback from the NCO.

The *FixPtGateway Out* blocks take the fixed-point value and convert it back to a double floating-point representation so that it can be graphed versus time in the *Scope* block.

Figure A2 is the phase detector block structure embedded in Figure A1. The three input lines are seen at the left of the figure. Inputs *In1* and *In3* are the sampled signal and immediately go into a *FixPtGateway In* block that converts the floating-point values to a fixed-point representation. The final input, *In2*, is the total phase of the NCO. It serves as input to a block labeled *sin/cos*. This block takes the phase and by use of a look-up table, outputs the sine and cosine of the input phase. The output, *Out1*, of the PD is the phase error of the system plus higher frequency terms.

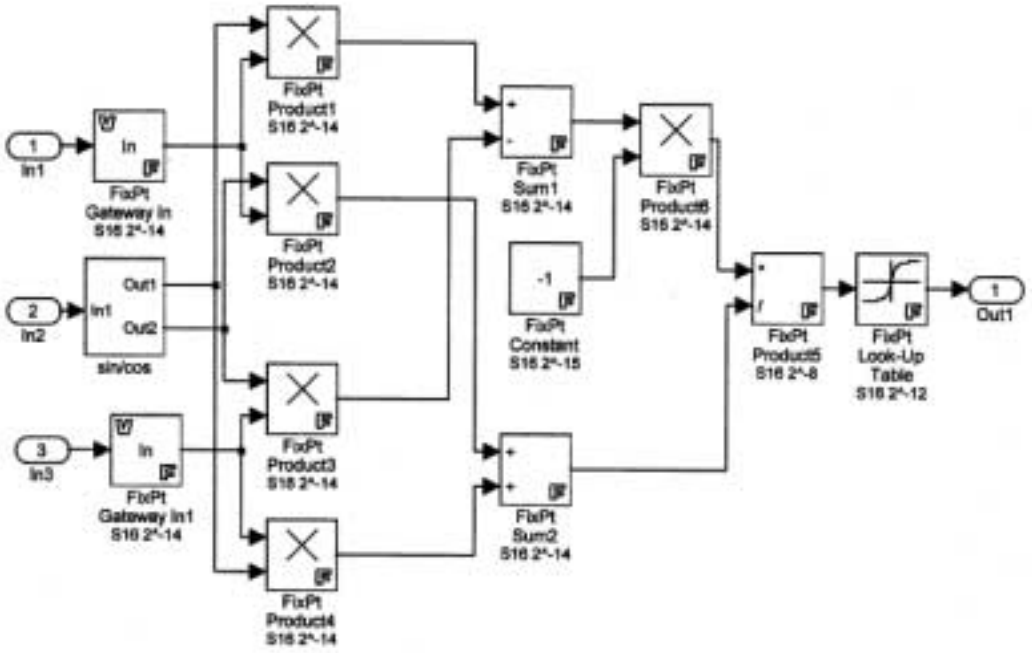

Figure A2. Phase Detector Simulink Model

The next block from Figure A1 is the loop filter. This block is shown in detail in Figure A3. The constants 635.146 and –621.44 are the lead-lag constants b_0 and b_1. The 1/z blocks signify a unit time delay. This block saves the output from the last sample time and outputs it at the current time sample. The output of this component is the phase error of the system.

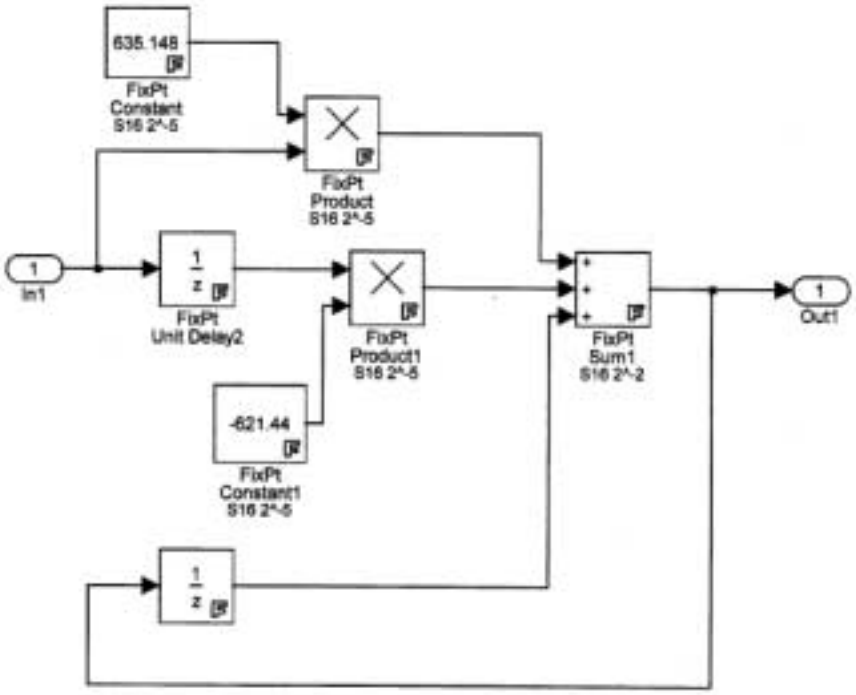

Figure A3. Simulink Active P-I Loop Filter

The final component block of Figure A1 is the NCO. This subsystem is shown in Figure A4. It takes as input the error from the loop filter and calculates the total phase error of the system. This phase error is fed back to the phase detector, and the process repeats until the error is forced to zero. The complicated circuitry at the end of the NCO is a comparator and switch combination that compares the phase output to π. If the output is less than π, the switch outputs its top input, which is just the signal. If the phase is greater than π, the switch outputs the bottom input, which is the signal minus 2π. This circuitry keeps the phase from growing unbounded or out of range of the fixed-point model.

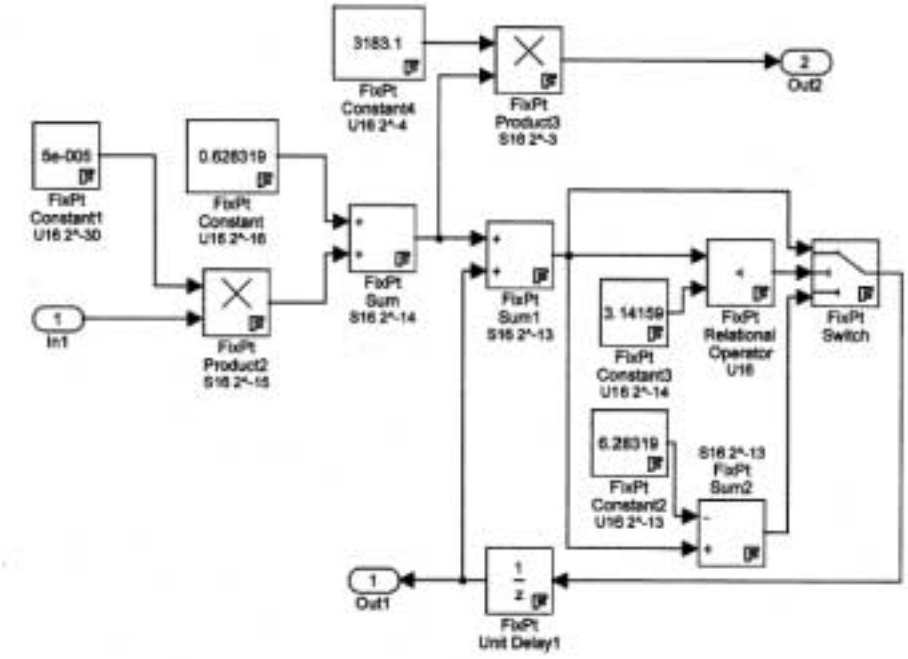

Figure A4. Simulink Numerically Controlled Oscillator

THIS PAGE INTENTIONALLY LEFT BLANK

LIST OF REFERENCES

[1] Best, Roland E., *Phase-Locked Loops*, 4th ed., McGraw Hill Companies, 1999.

[2] Gardner, Floyd M., *Phaselock Techniques,* 2d ed., John Wiley and Sons, New York, 1979.

[3] McCloskey, James, "Application of VHDL to Software Radio Technology," presented at the *IEEE IVC/VIUF1998 Conference in Santa Clara, CA,* March 1998.

[4] Proakis, John G. and Manolakis, Dimitris G., *Digital Signal Processing, Principals, Algorithms, and Applications,* 3d ed., Prentice Hall, Inc. Upper Saddle River, NJ., 1996.

[5] Sanneman, R.W., and Rowbotham, J.R., "Unlock Characteristics of the Optimum Type II Phase-Locked Loop," *IEEE Trans. Aerosp. Navig. Electron.,* vol. ANE-11, March 1964, pp. 15-24.

[6] Rappaport, Theodore S., *Wireless Communications, Principles and Practice,* 2d ed., Prentice Hall, Inc., Upper Saddle River, NJ., 2002.

[7] Navabi, Zainalabedin, *VHDL*, 2d ed., McGraw-Hill Companies, New York, NY, 1998.

[8] Bose, Vanu G., "The Impact of Software Radio on Wireless Networking," *Mobile Computing and Communications Review,* Vol. 3, No 1., January 1999.

[9] Egan, William F., *Phase-Lock Basics,* John Wiley and Sons, Inc. New York, NY, 1999.

[10] Gordon, Robert, "A Calculated Look at Fixed-Point Arithmetic," presented at *Communications Design Conference,* San Jose, CA., September 1998.

[11] Klopfer, Ron, "DSP-Based Digital Radio Design," *Communications Systems Design Magazine,* October 1998, http://www.csdmag.com/main/9810/9810feat3.htm.

[12] Marguiles, Allan S. and Mitola, Joseph III., "Software Defined Radios: A Technical Challenge and a Migration Strategy," *IEEE* 5th International Symposium, Volume: 2, pp. 551-556, 1998.

[13] Nise, Norman S., *Control Systems Engineering,* 3d ed., John Wiley and Sons, Inc. New York, NY, 2000.

[14] Wolaver, Dan H., *Phase-Locked Loop Circuit Design,* Prentice Hall, Inc., Englewood Cliffs, NJ., 1991.

[15] *Fixed-Point Blockset User's Guide,* ver. 3., The Mathworks Inc., 2000.

[16] *Real-Time Workshop User's Guide,* ver. 4., The Mathworks Inc., 2000.

[17] www.wwnet.net/~stevelim/fixed.html, July 2002.

THIS PAGE INTENTIONALLY LEFT BLANK

INITIAL DISTRIBUTION LIST

1. Defense Technical Information Center
   Ft. Belvoir, Virginia

2. Dudley Knox Library
   Naval Postgraduate School
   Monterey, California

3. Marine Corps Representative
   Naval Postgraduate School
   Monterey, California

4. Director, Training and Education, MCCDC, Code C46
   Quantico, Virginia

5. Director, Marine Corps Research Center, MCCDC, Code C40RC
   Quantico, Virginia

6. Marine Corps Tactical Systems Support Activity (Attn: Operations Officer)
   Camp Pendleton, California

7. Chairman, Coed EC
   Department of Electrical and Computer Engineering
   Naval Postgraduate School
   Monterey, California

8. Professor Tri T. Ha
   Department of Electrical and Computer Engineering
   Naval Postgraduate School
   Monterey, California

9. Professor Roberto Cristi
   Department of Electrical and Computer Engineering
   Naval Postgraduate School
   Monterey, California